A Formal Framework for Specification-Based Embedded Real-Time System Engineering

by

Martin Ouimet

B.S.E., Princeton University (1998)
S.M., Massachusetts Institute of Technology (2004)

Submitted to the Department of Aeronautics and Astronautics in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2008

Author ....
Department of Aeronautics and Astronautics
May 23rd, 2008

Certified by ....
I. Kristina Lundqvist
Professor, Mälardalen University
Thesis Advisor

Certified by ....
R. John Hansman
Professor, Massachusetts Institute of Technology
Doctoral Committee Chair

Accepted by ....
David L. Darmofal
Chairman, Department Committee on Graduate Students

Certified by . . . . [illegible]

Nancy A. Lynch
Professor, Massachusetts Institute of Technology
Thesis Supervisor

Certified by . . . . [illegible]

Heidi C. Perry
Director, The Charles Stark Draper Laboratory
Thesis Supervisor

Certified by . . . . [illegible]

Nicholas Roy
Professor, Massachusetts Institute of Technology
Thesis Supervisor

A Formal Framework for Specification-Based Embedded Real-Time System Engineering

by

Martin Ouimet

Submitted to the Department of Aeronautics and Astronautics
on May 23rd, 2008, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

The increasing size and complexity of modern software-intensive systems present novel challenges when engineering high-integrity artifacts within aggressive budgetary constraints. Among these challenges, ensuring confidence in the engineered system, through validation and verification activities, represents the high cost item on many projects. The expensive nature of engineering high-integrity systems using traditional approaches can be partly attributed to the lack of analysis facilities during the early phases of the lifecycle, causing the validation and verification activities to begin too late in the engineering lifecycle. Other challenges include the management of complexity, opportunities for reuse without compromising confidence, and the ability to trace system features across lifecycle phases. The use of models as a specification mechanism provides an approach to mitigate complexity through abstraction. Furthermore, if the specification approach has formal underpinnings, the use of models can be leveraged to automate engineering activities such as formal analysis and test case generation. The research presented in this thesis proposes an engineering framework which addresses the high cost of validation and verification activities through specification-based system engineering. More specifically, the framework provides an integrated approach to embedded real-time system engineering which incorporates specification, simulation, formal verification, and test-case generation. The framework aggregates the state-of-the-art in individual software engineering disciplines to provide an end-to-end approach to embedded real-time system engineering. The key aspects of the framework include:

- A novel specification language, the Timed Abstract State Machine (TASM) language, which extends the theory of Abstract State Machines (ASM). The TASM language is a literate formal specification language which can be applied and multiple levels of abstraction and which can express the three key aspects of embedded real-time systems – function, time, and resources.

- Automated verification capabilities achieved through the integration of mature analysis engines, namely the UPPAAL tool suite and the SAT4J *SAT* solver. The verification capabilities provided by the framework include completeness and consistency verification, model checking, execution time analysis, and resource consumption analysis.

- Bi-directional traceability of model features across levels of abstraction and lifecycle phases. Traceability is achieved syntactically through archetypical refinement types; each refinement type provides correctness criteria, which, if met, guarantee semantic integrity through the refinement.

- Automated test case generation capabilities for unit testing, integration testing, and regression testing. Unit test cases are generated to achieve TASM specification coverage through the rule coverage criterion. Integration test case generation is achieved through the hierarchical composition of unit test cases. Regression test case generation is achieved by leveraging the bi-directional traceability of model features.

The framework is implemented into an integrated tool suite, the TASM toolset, which incorporates the UPPAAL tool suite and the SAT4J *SAT* solver. The toolset and framework are evaluated through experimentation on three industrial case studies – an automated manufacturing system, a "drive-by-wire" system used at a major automotive manufacturer, and a scripting environment used on the International Space Station.

Thesis Supervisor: I. Kristina Lundqvist
Title: Professor, Mälardalen University

Thesis Supervisor: R. John Hansman
Title: Professor, Massachusetts Institute of Technology

Thesis Supervisor: Nancy A. Lynch
Title: Professor, Massachusetts Institute of Technology

Thesis Supervisor: Heidi C. Perry
Title: Director, The Charles Stark Draper Laboratory

Thesis Supervisor: Nicholas Roy
Title: Professor, Massachusetts Institute of Technology

Acknowledgments

First and foremost, I would like to thank my advisor and thesis supervisor, Professor Kristina Lundqvist, for providing me with the guidance and with all the resources that I needed to succeed. I have a tremendous amount of admiration and respect for Professor Lundqvist as an intellectual leader, as a mentor, and as a friend. I would also like to thank Professor John Hansman for accepting the role of doctoral committee chair toward the end of my studies. Other members of my committee also deserve praise - Professor Nancy Lynch, Professor Nicholas Roy, and Heidi Perry - for taking time out of their busy schedules to guide me through the PhD process. Your time and feedback was invaluable. Professor Daniel Jackson also deserves mention due to his involvement in the general examination process.

On a research note, I would like to acknowledge JK Srinivasan, Professor Mikael Nolin, Professor Egon Börger, Gustaff Naeser, Yuri Gurevich, David Wang, Guillaume Berteau, and Matthieu Quenot, who have contributed directly and indirectly to the research included in this thesis. Furthermore, the Charles Stark Draper Laboratory deserves a special mention, since it has provided the funding for the project. Special thanks go out to Lauren Kessler and Heidi Perry who were instrumental to make this project a reality. I would also like to recognize Professor Charles Coleman for giving me the opportunity to explore various areas and for encouraging me to satisfy my intellectual curiosity.

On a personal note, I would like to give special consideration to my dear friends, who have withstood my intermittent emotional outbursts and my constant state of flux. Nicolas Dulac deserves special mention for being such a great friend. Michel Duranceau was also there when I needed him the most, even while 350 miles away. Étienne Parent was an awesome roommate and a true friend. Bart East and Jason Smyczek remained supportive, even though I wasn't always keeping in touch. Thank you to the East family and to the Duranceau family for continuing to be wonderful people. Yves Boussemart was always a source of positive attitude and occasional memorable incidents. Thank you to other friends - MV, TM, MAC, CG, HH, JL, LO, NP, DW, CL, EP, JP, CL, MF, ... The MIT hockey team provided much needed entertainment and good times during my first years at MIT. The music crew also played an important role in my intellectual and emotional development - Rusty Scott, Victoria Chang, Elaine Kwon, Mark Kroll, Curtis Hughes, Mark Harvey, Tony Savarino, and Yoko Miwa. I would also like to recognize the wonderful ladies of the Boston area and beyond who, on various occasions, provided much needed emotional support and fantastic companionship to discover the wonderful nightlife of Boston and beyond. At the forefront, Wendy and Stella greatly contributed to my general well-being. WX, SJ, JQ, NT, MT, YJH, SSC, JC, AC, RC, V, LR, SD, CY, MM, MN, ... thanks for putting up with me, you are all awesome. Another thank you goes out to the fantasy baseball crew of New Jersey and Laval and to the fantasy hockey crowd of Montréal. The "Semi-Pro" crew, led by my uncle Michel, also deserves a wink.

Finally, a special "thank you" goes to my family who, at times, seemed more worried than my advisor about whether or not I would complete the PhD program. I have always maintained that if I could have half the heart of my father and half the courage of my mother, I would be extremely proud of who I am. Hopefully, I will get there some day. Thank you Gérard and Denise, you mean more to me than I can convey in a few lines. Thank you also to my brother, Patrick, my sister-in-law Annie, and their two sons Félix and Renaud. Your continuous support and distractions continue to be indispensable. Thank you also to my cousins Daniel and Richard. I promise that some day we will finally make it to Fenway Park. I am blessed to have a large extended family who is always radiant. Since they are too numerous to mention, I will not single them out individually; but kisses and firm handshakes go out to all the aunts, uncles, and cousins on both sides of my family.

Dear mom and dad, although I will probably never stop being a student in the figurative sense, I am convinced that the next time an acquaintance asks you what your younger son does for a living, you will be extremely happy that you will no longer have to answer "student". Warm regards to all who helped make this a reality. With love,

Martin

Remerciements

D'abord et avant out, j'aimerais remercier mon superviseur et ma directrice de thèse, le Professeur Kristina Lundqvist, qui m'a guidé dans ma recherche et qui m'a fourni toutes les ressources nécessaires pour assurer mon succès. J'ai énormément d'admiration et de respect pour le Professeur Lundqvist en tant que modèle intellectuel, en tant que conseillère, and en tant qu'amie. Je voudrais aussi remercier le Professeur John Hansman, qui a accepté le rôle de chaire du comité d'évaluation durant la fin de mes études. Je me dois aussi de mentionner les autres membres de mon comité - le Professeur Nancy Lynch, le Professeur Nicholas Roy, et Heidi Perry. Merci d'avoir pris le temps et d'avoir eu la patience pour me guider à bon port. Vos efforts et votre attention ont été grandement appréciés. Le Professeur Daniel Jackson mérite également mes remerciements dû à sa participation à l'examen général.

Du côté de la recherche, je voudrais créditer JK Srinivasan, le Professeur Mikael Nolin, le Professeur Egon Börger, Gustaff Naeser, Yuri Gurevich, David Wang, Guillaume Berteau, et Mathieu Quenot, qui ont tous contribué a la recherche inclue dans cette thèse. De plus, le Charles Stark Draper Laboratory obtient une mention spéciale, puisqu'il a subventionné ma recherche au cours de mes études au doctorat. Un gros merci à Lauren Kessler et à Heidi Perry pour leur dévouement. Je voudrais aussi remercier le Professeur Charles Coleman qui m'a donné l'opportunité d'explorer mes intérêts à ma guise et qui m'a encouragé à satisfaire ma curiosité intellectuelle.

Du côté personnel, j'aimerais offrir ma profonde gratitude a mes chers amis, qui, à un moment ou un autre, ont tous dû subir les répercussions de mes sautes d'humeur et de mon incertitude constante. Nicolas Dulac a été de loin mon meilleur ami et compagnon spirituel à Boston. Même s'il était à 350 miles de Boston, mon meilleur ami de longue date, Michel Duranceau, a toujours su être là pour me remonter le moral. Étienne Parent a été un ami exemplaire avec qui j'ai pu chialer plus souvent qu'à mon tour. Bart East et Jason Smyczek ont su apporter leur support, même si je n'ai pas toujours été aussi communicatif que j'aurais dû l'être. Un gros merci à la famille East et à la famille Duranceau qui continuent d'être des gens extrêmement chaleureux.

Yves Boussemart a toujours été une source d'énergie positive et d'occasionnels incidents mémorables. Merci aux autres amis - MV, TM, MAC, CG, HH, JL, LO, NP, CL^2, EP, JP, DMW, ... J'ai également découvert la musique lors de mon séjour aux études - Rusty Scott, Victoria Chang, Elaine Kwon, Mark Kroll, Curtis Hughes, Mark Harvey, Tony Savarino, et Yoko Miwa - merci d'avoir été une source d'inspiration, merci de votre enseignement, et merci de votre encouragement. Je voudrais aussi mentionner ces divines demoiselles de Boston et d'ailleurs, qui ont su enflammer plus d'une soirée, tout en apportant un support émotionnel indispensable et une complicité remarquable, et avec lesquelles j'ai pu découvrir différents restos et bistrots de Boston et d'ailleurs. En tête de liste, Wendy et Stella ont grandement contribué a mon bien être. WX, SJ, JQ, NT, MT, YJH, SSC, JC, AC, RC, V, LR, SD, CY, MM, MN, ... merci de m'avoir toléré, vous êtes sublimes. Je tiens aussi à remercier la gang du pool de baseball du New Jersey et les gangs des pools de hockey et de baseball de Laval. Clin d'oeil aussi a la gang du "Semi-Pro", avec mon oncle Michel en tête.

Finalement, j'offre un remerciement spécial a ma famille, qui m'a supporté moralement et financièrement tout au long de mes études. J'ai toujours cru que si j'avais la moitié du coeur de mon père et la moitié du courage de ma mère, je serais très fier de l'homme que je suis devenu. J'espère y arriver un jour. Merci Gérard et Denise, vous êtes plus importants pour moi que je ne puisse le témoigner en quelques lignes. Merci aussi à mon frère Patrick, à ma belle-soeur Annie, et à mes deux neveux Félix et Renaud. Merci aussi á mes cousins Daniel et Richard. Je vous promets qu'on ira au Fenway Park éventuellement. Je me compte très chanceux d'être membre d'une grande famille étendue. Malheureusement, ils sont trop nombreux pour être nommés individuellement; tout de même, j'offre baisers et poignées de mains viriles à toutes mes tantes, à tous mes oncles, et à tous mes cousins et cousines. Chers môman et pôpa, même si je ne cesserai probablement jamais d'être un étudiant au sens figuré, je suis convaincu que la prochaine fois où une connaissance vous demandera ce que votre plus jeune fils fait dans la vie, vous serez très fiers de ne plus avoir à répondre "étudiant". Merci à tous ceux et celles qui ont contribué à mon succès. Avec amour,

Martin

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This chapter serves as an "executive summary" of the work presented in this thesis. The chapter contains information about the motivations for the presented research, the contributions of the research, the list of presentations, posters, technical reports and refereed publications related to the research, and the roadmap of the thesis. Each chapter in the thesis follows a template structure, as explained in Section 1.5.

1.1 Motivations

In modern society, software systems can be found everywhere, including in airplanes, in automobiles, and in consumer electronics. The proliferation of software increases the dependency on the correct functioning of software and yields a new set of challenges in the engineering of software-intensive systems. The growing size and complexity of modern software systems exacerbates the difficulty of delivering reliable systems within aggressive budgetary constraints. In various engineering disciplines, the use of models has proven a viable approach to mitigate complexity through abstraction [40]. However the use of models in the engineering of software is a relatively novel approach to building software systems. The use of models not only helps to control complexity, but if the models have formal underpinnings, the models can be used to automate certain engineering activities such as verification and test case generation.

The research presented in this thesis seeks to address five key challenges in the engineering of embedded real-time systems:

- The high complexity of modern software systems, by providing a model-based approach to software-intensive system engineering.
- The high cost of Verification and Validation (V & V) activities by leveraging the use of models to automate engineering activities.
- The challenges in using formal methods in an engineering context by providing a novel literate specification language and abstracting verification details in a push-button approach.
- The lack of integration between disparate models by providing bi-directional traceability across levels of abstraction.
- The lack of integration of the state-of-the art in individual disciplines by providing an overarching engineering framework.

These five challenges form the base motivation for the research presented in this thesis. These challenges are addressed individually in subsequent chapters. In addressing these challenges, the research presented in this thesis makes a number of research contributions in various areas. These contributions are outlined in the following section.

1.2 Thesis Contributions

The research presented in this thesis makes five key contributions to address the challenges enumerated in the previous section:

- A new specification language for embedded real-time systems, the Timed Abstract State Machine (TASM) language, which extends the theory of Abstract State Machines (ASM). The TASM language integrates the specification of functional and non-functional properties - function, time, and resources.

- A set of verification procedures for automated analysis of models, using generally available analysis engines. The procedures include the analysis of completeness and consistency, the analysis of execution time, and the analysis of resource consumption.

- An approach to traceability of system models that incorporates syntactic change and semantic integrity.

- A generic and extensible approach to automatically generate test cases for unit testing, integration testing, and regression testing.

- An integrated framework for modeling, simulation, verification, and test-case generation for embedded real-time systems.

- An integrated toolset implementing the capabilities of the framework.

1.3 Relevant Publications

The research presented in this thesis has led to presentations and poster sessions presented at the "Real-Time System Symposium (RTSS)" in December 2006 [189], at the "ARTIST Workshop on Tool Platforms for Embedded System Modeling, Analysis and Validation" of the "Computer-Aided Verification Conference (CAV)" in July 2007 [196], at the "Real-Time in Sweden Conference (RTiS)" in August 2007 [197], and at the "Asia-Pacific Software Engineering Conference (APSEC)" in December 2007 [204].

The presented research has also yielded a number of Technical Reports released through the Embedded Systems Laboratory (ESL) at the Massachusetts Institute of Technology [190, 195, 201, 202, 206]. A reference manual for the Timed Abstract State Machine (TASM) language [185], as well as a user guide for the TASM toolset [186] are also available through the Embedded Systems Laboratory [88].

The Timed Abstract State Machine language was presented to the real-time system community at the "Real-Time and Network Systems Conference (RTNS)" in

March 2007 [199] and to the ASM community at the "Abstract State Machines Workshop (ASM)" in June 2007 [198]. A journal article about the TASM language is set to appear in Volume 14 of the Journal of Universal Computer Science (JUCS) in July 2008 [205]. A critical look on how time is treated in modeling languages, motivating the TASM approach to modeling time, was presented at the "Modeling in Software Engineering (MiSE) Workshop" of the "International Conference on Software Engineering (ICSE)" in May 2007 [194].

The case study involving the Electronic Throttle Controller (ETC) was presented at the "Critical System Development using Modeling Languages Workshop (CS-DUML)" of the "Model Driven Engineering Languages and Systems Conference (MoDELS)" in October 2006 [187] and was selected as one of two best papers to appear in Volume 4364 of Lecture Notes in Computer Science (LNCS) entitled "Models in Software Engineering" [188]. The analysis of TASM models using a *SAT* Solver was presented at the "Model-Based Testing Workshop (MBT)" of the "European Joint Conferences on Theory and Practice of Software (ETAPS)" in May 2007 [192] and appears in Volume 190 of "Electronics Notes in Theoretical Computer Science (ENTCS)" [193]. The TASM toolset was presented at the "Computer-Aided Verification Conference (CAV)" in July 2007 [200].

An overview of the framework will be presented at the "International Symposium on Quality Engineering for Embedded Systems (QEES)" [203], a symposium held jointly with the "European Conference on Model Driven Architecture Foundations and Applications (ECMDA)", in June 2008. A follow-up article will be submitted to the journal entitled "Software Tools for Technology Transfer".

1.4 Thesis Outline

This section provides an overview of the content of each chapter contained in this thesis.

- **Chapter 1: Introduction**

This Chapter provides an "executive summary" of the thesis and should be read before other chapters.

- **Chapter 2: Background Information**

  This chapter provides background information necessary to understand the material contained in the research. This chapter covers a wide range of topics such as information about real-time systems, software engineering, and descriptions of the analysis engines used to implement the framework. The reader is invited to browse sections of this chapter as needed.

- **Chapter 3: Framework Overview**

  This chapter provides an overview of the capabilities of the framework as well as the tool architecture used in the implementation of the framework.

- **Chapter 4: The Timed Abstract State Machine Specification Language**

  This chapter describes the Timed Abstract State Machine (TASM) Language, its syntax, semantics, and modeling facilities, including how time and resources are treated, hierarchical composition, and parallel composition. Throughout the chapter, illustrative examples are provided to depict the concepts as they are introduced.

- **Chapter 5: Static Analysis**

  This chapter presents the types of analysis that can be performed in the framework. The analysis procedures include completeness and consistency analysis, execution time analysis, and resource consumption analysis. This chapter also contains illustrative examples to demonstrate the analysis algorithms. The implementation of the analysis facilities, performed in the TASM toolset, are also described.

- **Chapter 6: Bi-Directional Traceability**

This chapter explains the bi-directional traceability capabilities of the proposed framework. The approach to traceability establishes a relationship between two or more TASM models and combines syntactic change management with notions of semantic equivalence for the models.

- **Chapter 7: Test Case Generation**

  This chapter presents the automated test case generation capabilities of the framework for unit, integration, and regression test case generation. The regression test case generation strategy uses the traceability approach described in Chapter 6. Examples and implementation details are also presented.

- **Chapter 8: Case Studies**

  This chapter contains the results of the three case studies used to evaluate the research presented in this thesis – the production cell case study, an Electronic Throttle Controller (ETC), and the Timeliner Script Executor. The case studies are introduced in Section 2.8 but the models and analysis results are presented in Chapter 8. The complete TASM models for each case study are provided in the appendices.

- **Chapter 9: Conclusion**

  This chapter provides a critical evaluation of the presented research, draws conclusions for the thesis, and describes directions for possible future developments of the research.

- **Appendix A: TASM Language Reference**

  This appendix contains the concepts involved with the implementation of the TASM language in the TASM toolset. These concepts include the complete Context-Free Grammar (CFG) for the TASM language and various implementation topics such as operator precedence and typing.

- **Appendix B: Translating TASM Models to** ***SAT***

This appendix contains the details of the mapping from the TASM language to *SAT* , for the purpose of static analysis, as explained in Chapter 5, and for test case generation, as explained in Chapter 7.

- **Appendix C: Translating TASM Models to UPPAAL**

  This appendix contains the complete mapping from the TASM language to UPPAAL , for the purpose of timing analysis, as explained in Section 5.3. The mapping to UPPAAL is also used for model checking of functional properties.

- **Appendix D: Production Cell TASM Model**

  This appendix contains the complete TASM model for the production cell case study, explained in Section 2.8.1 and studied in Section 8.1.

- **Appendix E: Electronic Throttle Controller TASM Model**

  This appendix contains the complete TASM model for the electronic throttle controller for the case study, explained in Section 2.8.2 and studied in Section 8.2, in Section 8.3, and in Section 8.4.

- **Appendix F: Timeliner Plant Simulator TASM Model**

  This appendix contains the complete TASM model for the Timeliner case study involving the plant control system, explained in Section 2.8.3 and studied in Section 8.5.

1.5 Chapter Structure

Individual chapters in this thesis follow a common template. At the beginning of each chapter a paragraph explains the information contained in the chapter. The last section of each chapter is named "Segue into Chapter N" and provides a brief summary of the information provided in the current chapter and how it leads to the following chapter. All chapters follow this template except for the appendices. Unlike the thesis chapters, individual appendices do not follow a common pattern. However, the first paragraph of each appendix contains a brief description of its content. The content of the appendices is summarized in Section 1.4. Appendices do not follow a linear progression and serve as a reference that can be read in any order.

1.6 Notational Conventions

In order to enhance the readability of the thesis, special fonts are used to clarify meaning in certain situations:

- *Italic font* is used for definitions and when referring to abstract syntax, such as the names of machines or the names of rules.

- `Teletype font` is used when referring directly to the concrete syntax of a model, such as a `variable`.

- "Quotation marks" are used to emphasize blocks of text that should be grouped together.

Furthermore, the listings for models expressed in the Timed Abstract State Machine (TASM) language are expressed in `teletype font`.

1.7 Segue into Chapter 2

This chapter presented an "executive summary" of the content of the thesis. The following chapters expand on this summary. The next chapter, Chapter 2, provides background information about the topics covered throughout the thesis.

Chapter 2

Background Information

This chapter presents background information related to the concepts explained in the following chapters. The information contained in this chapter includes details about the types of systems targeted by the presented research, details about how the current work integrates into software engineering practice, information about the analysis engines used in the framework, and descriptions of the case studies used to evaluate the presented framework.

2.1 Real-Time Embedded Systems

The primary goal of a computer system is to provide value, as defined by a user of the system, by performing a set of functions. More specifically, a computer system must provide a *correct* response (output) based on a given stimulus (input). The concept of *correctness* is comprised of multiple facets such as functional correctness and non-functional correctness. The *functional correctness* of a computer system is defined through a set of *requirements* that the input-output behavior of the system must satisfy [85]. The requirements describing the functional correctness of a computer system can be described through *safety properties*, that is, statements about behavior that *should never occur*, and *liveness properties*, that is, statements about behavior that *should eventually occur* [30]. All functional behavior of computer systems can be described in terms of safety and liveness properties [154].

While functional correctness is a critical aspect of all computer systems, certain types of systems also require that non-functional aspects of the system, such as timing behavior, conform to stringent correctness criteria. The research presented in this work addresses the engineering of embedded real-time systems, a class of systems where non-functional properties are central to the system's value. More specifically, embedded real-time systems represent a special class of computer systems where time plays a critical role in the correctness of the system. In a *real-time* system, the system must not only produce a correct answer, but must also do so in an adequately bounded amount of time [58]. The amount of time under which the system must produce a response is termed a *deadline*. If a real-time system provides an answer after a deadline has elapsed, the system is said to have *missed a deadline*. With regards to correctness, the implications of missing a deadline depends on the type of system. Real-time systems fall into two categories – *hard* real-time systems, where missing a deadline is unacceptable, and *soft* real-time systems, where missing a deadline may be acceptable under certain circumstances, depending on performance requirements [155]. Nevertheless, time plays a critical role in defining the correctness of a real-time system since a correct answer provided too late can be as erroneous as providing an incorrect answer [49]. The timing analysis provided by the framework does not make assumptions about whether the system being analyzed is hard or soft. The framework provides generic timing analysis to determine the best and worst case timing behavior, and it is up to the system behavior to decide whether the analysis results are acceptable for the system being engineered.

In practice, real-time systems are also typically *reactive*, meaning that they do not terminate, but are in continuous interaction with the environment, until the system is switched off. Reactive systems are different than *transformational* system, where the system terminates after producing an answer. In a reactive real-time system, the timing correctness of the system is defined as the absence of missed deadlines while taking into account the continuous interaction of the system with its environment. The types of systems targeted by the proposed framework are of the *mostly periodic* nature [164], meaning that they operate in a continuous loop that samples the environment through sensors, makes a decision on what action the system should take based on the sensor values, and affects the environment by executing the action through an actuator. A sample loop for the systems targeted by the proposed framework is shown in Figure 2-1. It is important to note that which steps of the loop are executed at each iteration is implementation dependent. As will be described in the electronic throttle controller case study, the sampling of the state through sensors could be done at a lower frequency than, for example, the frequency of deciding on the action to be taken by the controller.

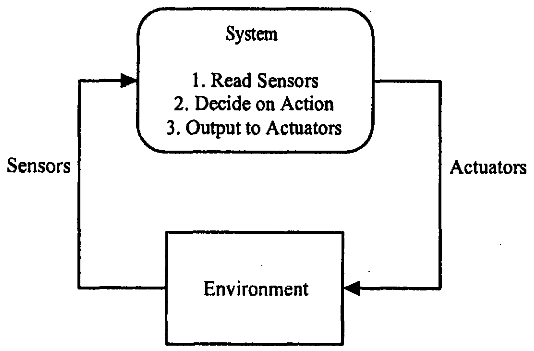


Figure 2-1: High level view of a mostly periodic reactive real-time system

In traditional real-time system theory, the concept of a deadline refers to the Worst-Case Execution Time (WCET) [89] also sometimes called the *worst-case computation time*, that is, the maximum time that can elapse when an individual task or an individual process executes [58]. In this research, the traditional WCET definition is made more general to include system properties. In the remainder of this work, the term WCET is used to denote the maximum amount of time that can elapse when the system completes a path between two states. The definition used in this thesis also stipulates that such a path can consist of *any* two states. In this definition, the notion of *state* can include both the state of the engineering artifact such as the program counter and values of system variables, *and*, the value of environment variables. This definition is more general than the traditional definition and can capture important concepts such as end-to-end latency all the while being able to express the traditional definition. *End-to-end latency* refers to the longest reaction time of a system to an environment stimulus, taking into account system properties such as environment interaction, task interference, and delay in response. For example, in the loop of Figure 2-1, the value of an environment variable could change while the system is deciding on which action to take. Depending on the frequency of sensor readings, a significant delay could result in the system taking a corrective action since there could be a delay before the change is detected. Figure 2-2 shows a symbolic view of the time that can elapse between an event occurring and the system taking a corrective action. The verification problem to ensure that there are no missed deadlines can be summarized as:

$$dt \ = \ t_2 \ - \ t_1 \ \leq \ \textit{Required Deadline}$$

The response latency, *dt*, refers to how much time elapses between the event and the response. The proposed framework provides necessary facilities to calculate the maximum value of *dt*, for any event and action modeled. The *Required Deadline* is system-dependent and is provided by the performance requirements. It is also the responsibility of the system designer to decide on which course of action to take if the designed system does not meet the required deadline. The proposed system provides only the necessary modeling and analysis facilities. Additional definitions related to execution time are given in Section 5.3.

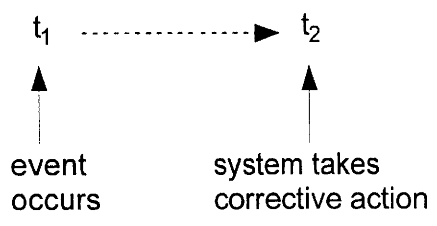


Figure 2-2: Delay in system responding to an event of interest

Most real-time systems also fall into the category of embedded systems. *Embedded* systems represent a special class of real-time systems where the software system is not stand-alone, but is part of a larger system and must work with other components to achieve the system's goals [155]. Vehicle controllers, such as automotive electronics and avionics, are typical examples of embedded real-time systems. In an embedded system, resources such as communication bandwidth and memory are typically limited and must be shared across multiple components. Consequently, the correctness of an embedded real-time system is also dependent on the resource usage being adequately bounded. In summary, for an embedded real-time computer system, the correctness of the system is defined in terms of three key aspects – functional correctness, timing behavior, and resource usage. These three aspects form the fundamental motivation of the modeling and analysis capabilities provided by the proposed framework.

2.1.1 The Nature of Time in Real-Time Systems

Since time plays an important role in defining the correctness of a real-time system, it is paramount to understand the role of time in the systems of interest. On a general and global level, the time axis is a monotonic function that is used to order events linearly according to some concept of progression [101]. A large body of research has been devoted to establish that a computer system satisfies a correct ordering of events [49]. This correct ordering of events, also called *qualitative time*, refers to the ordering of events with respect to one another and is not concerned with the temporal distance between events [212]. For example, in the well-known Simple Mail Transfer Protocol (SMTP), an acknowledgement message (ACK) shall not be received before a synchronization request (SYN) has been emitted. In other words, an ACK must occur after a SYN and never before. However, in real-time systems, timing correctness does not depend only on the ordering of events, but also depends on the numerical distance between events, a concept called *quantitative time* [155]. For example, in the SMTP protocol, after a SYN has been emitted, a timer is typically started while waiting for the ACK. If, after a prespecified amount of time, the ACK has not been received, the SYN sender might assume that the SYN request was lost. In such a situation, the precise amount of time between the SYN and the ACK is of particular importance, in addition to the messages occurring in the correct order.

Quantitative time appears in real-time system problems either explicitly or implicitly. Examples of where quantitative time appears explicitly include performance

Source	Example
Requirements	The data in the operator console shall be refreshed 10 times per second
Physics	It takes approximately 5 seconds for a projectile shot straight up in the air at a velocity of 50 m/s to come to rest at its apogee
Components	Pressure sensors can put data on the system bus at a rate of 10Hz

Table 2.1: Examples of sources of explicit quantitative time

requirements, local clocks, timeouts, scheduling, the physics of the problem, and constraints of the components of the system. Examples of explicit instances of quantitative time are shown in Table 2.1. Examples where time appears implicitly, as a side-effect, include software execution time and hardware execution time. Listing 2.1 shows a brief example of software code, written in the Timeliner scripting language [61], borrowed from the Timeliner case study. The code represents a sequence used to maintain cabin temperature between 20 and 25 Celsius degrees [238]. In order to determine how long this snippet of Timeliner code takes to execute, many other questions need to be answered. For example, the execution time of the script depends on:

- The properties of the execution platform
- The semantics of the language
- The assumptions on the behavior of the environment

Once the code has been written and the system is implemented, these concerns can typically be addressed to a satisfactory degree of confidence. For example, in [62], precise timing measurements of each statement of the Timeliner language have been measured through laboratory experiments for a specific execution environment. However, determining these execution times a priori remains a challenging endeavor.

Listing 2.1 Sequence TEMP_MONITOR [238]

```
SEQUENCE TEMP_MONITOR
  EVERY 1
    IF TEMPERATURE >= 26 THEN
      SET TRYING_TO_COOL_SYSTEM TO TRUE
      COMMAND COOLING, NEW_STATE=>ON
      WHEN TEMPERATURE <= 22
        SET TRYING_TO_COOL_SYSTEM TO FALSE
        COMMAND COOLING, NEW_STATE=>OFF
      END WHEN
    END IF
    IF TEMPERATURE <= 19 THEN
      COMMAND HEATING, NEW_STATE=>ON
      WHEN TEMPERATURE >= 22
        COMMAND HEATING, NEW_STATE=>OFF
      END WHEN
    END IF
  END EVERY
CLOSE SEQUENCE
```

Even for software and hardware execution, time can also appear implicitly and explicitly. For example, the code in Listing 2.1 contains one explicit timing statement, the "EVERY 1" statement. This statement tells the runtime system that the sequence shall execute at most once per second. Other examples of explicit timing statements include the statements *sleep* and *wait*, which are present in many programming languages such as C++ and Java [239]. In real-time system engineering, the explicit sources of quantitative time, outside of software and hardware, define the timing constraints of the system that is to be built. One of the goals of real-time system engineering is to build a system which meets these constraints.

2.2 Systems Engineering

Systems Engineering is the aggregation of multiple elements to perform functions that could not be performed by the elements alone [152]. Systems engineering is an overarching discipline which includes aspects bridging people, documentation, software, hardware, and other domains. Systems engineering efforts seek to develop processes, tools, and techniques to ensure that a given engineering artifact satisfies all parties involved throughout the lifetime of a system. In Section 2.1, the types of systems targeted by the presented framework were presented, alongside definitions of the correctness of these systems. The goal of the systems engineering efforts go beyond the correctness aspects described in Section 2.1 and include concepts related to the stakeholders, potential risks, safety concerns, and other factors affecting the engineering, delivery, and operation of the system [166]. The goal of this section is to situate the applicability of the presented framework in the wider sphere of systems engineering.

Figure 2-3 shows the steps of a generic systems engineering process defined in [19]. The framework presented in this research is applicable in the software and digital hardware engineering facets of real-time systems, during the modeling phases and the integration phases depicted in Figure 2-3.

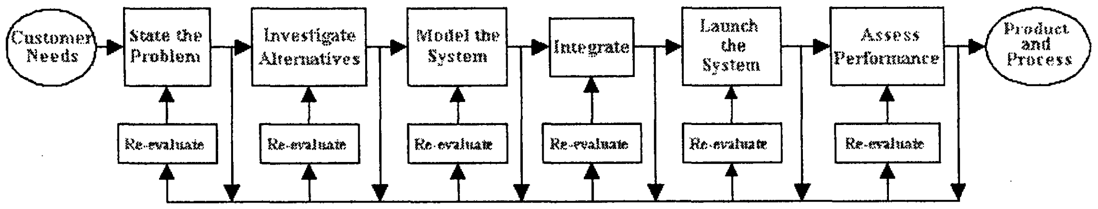


Figure 2-3: Systems engineering process [19]

The proposed framework assumes the existence of requirements on the functional, time, and resource aspects of the system. The engineering of software and digital hardware for an embedded system, such as the engineering of an avionics system, will be performed in parallel with other systems engineering activities such as requirements analysis and vehicle design. In the following section, software engineering principles are reviewed as motivations for the presented framework.

2.3 Software Engineering

Software engineering is the set of techniques, processes, and tools used to develop computer systems [242]. Typically, software engineering is divided into lifecycle phases that traditionally include requirements engineering, design, implementation, testing, and maintenance [255]. These different phases are carried out in sequence, with a certain amount of overlap depending on the lifecycle model that is used [253]. Validation and Verification (V & V) activities are defined as the process of establishing confidence into the correctness of the system. More specifically, *validation* refers to the activities carried out to ensure that the system being engineered will meet the user's needs. *Verification* refers to the activities carried out to ensure that the software behaves in accordance to the correctness criteria expressed as requirements. In practice, these activities comprise a mix of testing, user interviews, mathematical proofs, and various forms of human inspections [99]. V & V activities are typically the large ticket item on software engineering projects and can comprise over 50% of the development costs [39, 255]. The purpose of performing V & V activities is to eradicate defects from the system being engineered. A *defect* is a facet of the system which does not conform to the user's needs or to the required correctness criteria. V & V activities are typically carried out throughout all phases of the engineering lifecycle. Empirical evaluations of software engineering projects have demonstrated that the cost of finding and fixing a defect in a computer system increases dramatically the later it is found in the lifecycle [38, 39, 255]. Consequently, finding and fixing defects during the early phases of the engineering lifecycle can result in lower defect fixing costs and lower V & V costs. A typical software lifecycle consists of a number of phases that include requirements, design, implementation, testing, and maintenance [38]. A popular example of a popular lifecycle, the "V Lifecycle Model" [242], is depicted in Figure 2-4. In Figure 2-4, the engineering activities typically begin in the top left corner and proceed diagonally toward the bottom and back up toward the top right corner. However, software engineering lifecycle are typically iterative and hence do not follow a strict linear progression, indicated in Figure 2-4 by the arrows linking each phase.

Traditional approaches to software engineering have relied heavily on natural language documents and natural language communication to capture the requirements, design principles, and results of V & V activities relating to the system being engineered [71]. However, since natural language is ambiguous by definition, performing V & V activities based on natural language documents is error-prone and lack the type of repeatability that can be provided through automated analysis [225]. With the growing size and complexity of modern computer systems, relying solely on the intellectual rigor of engineers can lead to unpredictable results [124]. The framework

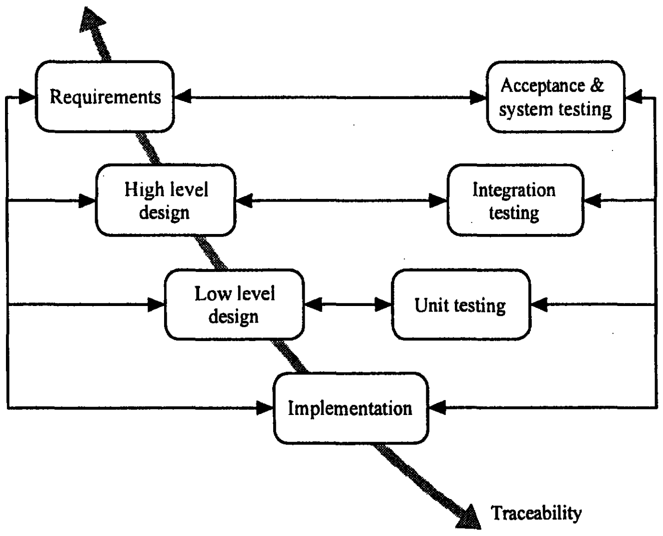


Figure 2-4: V software lifecycle model [242]

proposed in this research focuses on automating V & V activities, notably formal verification and test case generation, to provide a repeatable and reliable engineering approach that could lead to decreased V & V costs.

2.4 Formal Methods

Various efforts have attempted to remedy the shortcomings of natural language through the formalization of structured natural language [54, 73, 224, 246]. The need for a precise language and the benefits of automated analysis have motivated the development of specification approaches based on mathematics [71]. These attempts, also known as *formal methods* or *formal specification*, have yielded specification languages and proof systems that have a wide range of analysis capabilities, mostly through mathematical proofs. These approaches aim to address the lack of rigor of ad-hoc engineering techniques by rooting the engineering in well-founded mathematical principles. However, because many of these languages make use of advanced mathematics, they suffer from a lack of usability and readability without the proper expertise [51, 71]. The benefits and drawbacks of using formal methods have been documented heavily [50, 51, 71]. Cited benefits include the detection of defects early in the engineering cycle [118], precise and concise specifications [248], and the capability for automated analysis [220]. Cited drawbacks include the heavy use of arcane mathematical notations [71], the lack of scalability of most methods [50], and the large investment typically required to use formal methods [51]. Besides the negative connotation that the term *formal methods* has taken in some circles [124], the benefits of unambiguous specifications and the repeatability of automated analysis, throughout the phases of the lifecycle, have been generally accepted in the software engineering community [37, 179].

The challenges of engineering real-time systems has also led to various efforts to automate lifecycle activities. The lifecycle activities that can be automated include verification [70], validation [98], and test case generation [15, 222]. The automation of these activities is typically centered around a formal specification language or centered around a mathematical formalism. These approaches, also called *specification-based* development and *model-based* development, are finding increasing popularity within the industrial community and within the various research communities [179]. The terms *model-based* and *specification-based* are used interchangeably in the literature. In the context of this thesis, these terms are also indistinguishable, but the term *specification* is used to denote the description of behavior that can serve the dual purpose of the documentation of the intended system behavior and a model that can be analyzed. These approaches intersect with formal methods, under a different name, in that they rely on a notation with well-defined formal semantics. However, specification-based approaches focus on engineering activities and less on mathematical proofs, as is the case for formal methods [82]. In the model-based engineering domain, the research community has yielded a large body of languages, approaches, algorithms, and tools to specify, analyze, and automate engineering activities. While there have been key contributions in individual areas, it is currently challenging to incorporate the state-of-the-art in real-time system engineering into a cohesive framework [49] that can be used to engineer systems. These challenges are partly due to the lack of interoperability between existing approaches and tools [49, 179]. This lack of integration creates the need for engineering frameworks that integrate formal methods with specification concepts, such as the framework presented in this thesis. Furthermore, tool support, such as the capabilities provided by the proposed framework, also bridges the integration gap between modeling languages and formal analysis.

2.5 Model-Based Software Engineering

Model-based software engineering (MBSE), also called model-driven software engineering (MDSE) and model-driven system engineering, is an approach to software engineering where models play a central role in lifecycle activities [228]. The key point of the approach is the existence of a set of *models*, which are abstractions of the system to be implemented. The models contain information about the desired behavior of the system and are used to drive the lifecycle phases. Some of the benefits of having system models include the ability to simulate the prototype system and to perform analysis before implementation begins [257], leveraging the economics of software engineering to uncover defects as early as possible [38]. Model-based engineering approaches typically employ graphical or structured models that can be amenable to simulation and analysis, usually through a computer. As is the case for formal methods, models with well-defined semantics are means to remedy the ambiguity of natural language. A model-based approach is typically composed of a notation, formal or informal, used to express system behavior, and a set of associated methods and processes to ease engineering activities. The true benefits of a model-based approach occur when a *literate* [151] notation with formal semantics is used, so that the models can serve the dual purpose of an analysis mechanism and of the documentation of intended system behavior [146]. A *literate* specification language is a language which can be read like the English language and does not contain extraneous symbols aside from basic operators from arithmetic [151]. Furthermore, given the investment required to build models, the ability to automate engineering activities, such as test case generation, can help alleviate the cost of building and maintaining models [173].

Among the proponents of model-based software engineering, two professional organizations have proposed standards for the language and tools to be used for MBSE. The Object Management Group (OMG) has drafted a set of standards to enable model-driven software engineering, especially in the presence of disparate tools [179]. The purpose of the standard is to define information exchange formats so that various models and tools can be incorporated. OMG's efforts have been focused on using the Unified Modeling Language (UML) [181] as the underlying language of its model-driven efforts. The UML language relies heavily on object-oriented design approaches and has yet to adopt a standard formal semantics [138]. The Society of Automotive Engineers (SAE) preaches a similar model-based approach through the use of the Architecture and Analysis Design Language (AADL) [223], targeted at embedded real-time systems. The AADL language is an Architecture Description Language (ADL) that can be used to express high level component interaction and information flow. However, at the time of the writing of this thesis, AADL does not contain facilities for specifying component-level behavior.

2.6 Modeling Languages

In order to perform model-based software engineering, models must be expressed using a suitable modeling language. Section 2.1 establishes the correctness criteria of real-time systems, namely function, time, and resources. Furthermore, Section 2.1.1 describes how time is reflected in the engineering of real-time systems. In particular, Listing 2.1 shows an example of software code which expresses explicit and implicit timing behavior. In modeling languages, quantitative time concepts are almost always explicit [144]. The type of modeling addressed in this research is *behavioral* modeling, to capture the dynamic aspects of the system. Behavioral modeling is in contrast to *structural* modeling, which captures the static aspects of the system, e.g., a class inheritance hierarchy or a multiplicity relationship [134]. In behavioral modeling, system dynamics are typically represented as some form of transition system where the system transitions from one state to another state based on a set of conditions [144]. Traditional languages to represent state transition systems include finite state automata [232] and statecharts [122]. For most modeling languages, untimed versions of the language exist and time was added as an extension of the language. This is the case for timed automata [5], time/timed Petri nets [60], timed process algebra [159], the Timed Abstract State Machine language (TASM) [199], and the real-time profile of the Unified Modeling Language (UML) [180].

While all of these languages have similarities, they also have significant differences in how they represent and handle time. The two main time models are *discrete* time and *continuous* or *dense* time. In a discrete time model, time progresses in fixed constant steps $dt \in \mathbb{N}^+$. In a continuous time model, time evolves continuously, and any time-related value is taken from the Reals domain ($t \in \mathbb{R}$). Languages also differ on how time evolves. Time can evolve either in states or during transitions. For example, time annotations can be added to Petri nets in places or in transitions or in both [60]. The difference lies in whether the subject of the description is the duration of an action or the awaiting of an event. An example of a light switch, modeled in the timed automata of UPPAAL [157] is shown in Figure 2-5. The model describes the behavior of a lamp in relation with possible user interactions [24]. If the lamp is off and the switch is pressed, the lamp will turn on to the low setting. If, after the light has been turned on, the switch is pressed again within 5 time units, the lamp increases its intensity to the bright setting. On the other hand, if the lamp is on and the switch is pressed again, but more than five time units have elapsed, the lamp turns off. This example illustrates a model that describes the passage of time between events. In this model, events are instantaneous but the precise timing between events is of utmost importance.

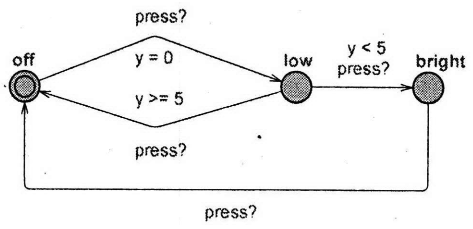


Figure 2-5: Timed automaton describing the behavior of a lamp [24]

Another way to represent time is to model events or actions as being durative instead of instantaneous. In the Timed Abstract State Machine language (TASM) [189], which will be presented in Chapter 4, time is attached to transitions to simulate durative actions. Listing 2.2 shows the actions of the robot of the production cell system [163], modeled in the TASM language. In the production cell problem, described in Section 2.8.1, a robot takes commands from a controller and executes these commands. When the robot is instructed to pick up a block, the action takes a certain amount of time to complete until the robot is available again to process other commands. In Listing 2.2, the action to pick up a block lasts 1 time unit.

Whether a language predominantly favors time passage or duration of actions in its notation is irrelevant from a pure expressivity perspective since both types of notations can be used to represent both concepts [30]. The differences lie in what paradigm better fits the problem being addressed. For the specification of real-time

Listing 2.2 Partial TASM model of a robot action to pick up a block

```
R1: Arm B at press, block is available -> pick up block
{
  t     := 1;
  power := 2000;

  if armbpos = atpress and armb = empty
     and press_block = available then
    press_block := notavailable;
    press       := empty;
    armb        := loaded;
}
```

systems, and for the modeling of software in general, the term *execution time* is used in numerous contexts. Most of the time, this term refers to the time to execute actions or, in other words, to the duration of actions. Verifying the correctness of a real-time system involves establishing that the durations of the actions of the system meet the time constraints of the requirements and of the problem domain.

2.6.1 The Time Paradox: Incorporating Time in High Level Models

Section 2.1.1 explained how time is reflected in real-time systems and Section 2.6 described how modeling languages express time. This subsection explains the paradox encountered when attempting to model system behavior that is closely tied to implementation details. In scheduling theory [63], the task graph [1] is the prevalent modeling formalism. A task graph is a directed graph where nodes represent tasks and edges represent precedence constraints between tasks. Each task is assigned an execution time, that is, a duration. A sample task graph with 7 tasks is shown in Figure 2-6. In Figure 2-6, each node represents a task and the numerical value next to the task represents the execution time of the task. Each arrow represents the precedence constraints, meaning that a task occurring at the beginning of an arrow must complete before the task at the end of the arrow can begin. The *scheduling problem* is concerned with finding a solution to scheduling the set of tasks on n processors, while enforcing the precedence constraints [57] and some notion of optimality.

Analogously, the *co-synthesis problem* concerns itself with optimal allocation of

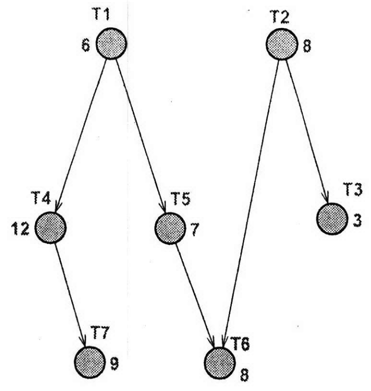


Figure 2-6: Sample task graph

a task graph to *processing elements* (e.g., reusable hardware (FPGA), application specific integrated circuits (ASIC), and software) [76]. The similarities between these two problems lie in the existence of a task graph, with known execution times for individual tasks. For the co-synthesis problem, this assumption seems misleading because the execution times will vary depending on which processing element a task is allocated to. On the other hand, for the scheduling problem, the task graph can be derived from an implementation. However, in real-time system engineering, the task graph is an abstraction of an implementation and, conceptually, should be defined before implementation begins. Defining the set of tasks and the dependencies between tasks should be a design decision, not an implementation one. Relying on the set of tasks to naturally emerge during coding causes development to remain an ad-hoc process at best, with little support for predictability. Furthermore, the scheduling problem also assumes that tasks have already been assigned to software, and therefore makes co-synthesis challenging. It is one of the goals of model-driven engineering to remedy ad-hoc system development by structuring engineering activities through the use of models. For real-time systems, can realistic models, such as task graphs, be built before implementing the system?

There are many possible answers to this paradox. Conceptually, design is and has always been an uncertain process where predictions that may or may not come true are made [56]. Nevertheless, design has proved to be a valuable activity in terms of cost and time saving, even in the face of uncertainty [37]. In a model-driven approach to development, it is highly unlikely that model-driven engineering will be a purely downstream activity flowing monotonically from model to implementation. More likely, feedback from downstream activities will be incorporated into upstream activities, leading to an iterative model-driven approach, where models are being adjusted as the implementation is being developed. Like any other topic in system engineering, experience with building models and experience with engineering using models will dictate the successful use of models in real-time system engineering. Moreover, rooting development around mature and predictable components, as is often the case in aerospace systems [234], greatly enhances the predictions that can be made by models.

At the modeling level, modeling notations are able to capture the uncertainty involved with annotating models with time. The use of interval semantics for durations gives a lower bound and an upper bound on durations. An example of a TASM specification with duration specified using interval semantics is shown in Listing 2.3.

Listing 2.3 TASM Model of an electronic throttle controller [187] (partial)

```
R1: Driving Mode
{
  t := [2, 5];

  if controller_mode = driving then
    throttle_v := Driving_Throttle_V();
}

R2: Limiting Mode
{
  t := [3, 8];

  if controller_mode = limiting then
    throttle_v := Limiting_Throttle_V();
}
```

Furthermore, the level of abstraction where the modeling occurs determines whether software times should be included in the model. For system models such as the production cell system [163], the physics of the problem and the time constraints on the system are on a scale much larger (on the order of seconds) than the time scale of the software (on the order of microseconds). Consequently, as it often happens in high level models, the software is fast enough given the problem definition and time does not need to be immediately included for software components in the models.

This is certainly the case in the production cell system where controller actions are approximated to be instantaneous [163].

A model-driven approach should have a notion of *refinement* [250], that is, a methodology to build models at different levels of abstraction, by gradually adding details to high level models. Furthermore, the refinement approach should have facilities to show a correspondence between two models at different levels of abstraction [17]. If such a notion is present, time estimates from high level models can become constraints on lower level models and, eventually, constraints on implementation. If an implementation cannot satisfy these constraints, the models will need to be adjusted in order to accommodate implementation characteristics. In this view, task graphs can be designed and approximated using high level models, making the scheduling problem and the co-synthesis problem relevant. During the design phase, analyzing schedulability and possible allocations to hardware and software can be useful to drive the implementation. In this research, the notion of *refinement*, as used in the formal methods community [79], is combined with the notion of *traceability*, as used in the system engineering community [217]. Traceability has traditionally been used to denote the ability to relate the syntax of disparate artifacts, including models, at different levels of abstraction. For example, in Figure 2-4, the traceability across lifecycle phases is depicted by the gray arrow on the left side of the figure. The benefits of traceability include the documentation of the dependency of various assumptions made throughout lifecycle phases [94]. However, traceability typically involves only the visualization of related artifacts and does not include notions of semantic equivalence that can be enforced through tool support. In contrast, notions of refinement in the formal methods community concern mostly only semantic equivalence between models and do not address the tracking of design assumptions [174]. Uniting these two notions, as performed in the proposed framework, combines the best of both worlds and provides a basis for end-to-end bi-directional traceability from high level models to implementation.

2.7 Analysis Engines

The growing need for more efficient software engineering has led to the development of sophisticated tools for computer-assisted analysis of software artifacts [124]. Among these analysis engines, theorem provers [84, 210], model checkers [30, 67], *SAT* solvers [175], Satisfiability Modulo Theory (SMT) Solvers [97, 229], and Linear Programming (LP) solvers [96, 218] have attracted large research efforts. All of these solvers support completely automated analysis, except for theorem provers. Because automation of engineering activities is a central goal of the proposed framework, theorem provers are not considered for the present version of the framework. Furthermore, the use of linear programming solvers and SMT solvers are treated as part of Future Work, in Section 9.3. The types and specific instances of analysis engines that are used in the presented research are explained in this section.

In the formal verification realm, model checkers [70] and *SAT* solvers [175] have been used to perform various types of analysis [140]. The popularity of model checkers and *SAT* solvers can be attributed to the full automation capabilities of the analysis, combined with the automated generation of a counterexample when a property to be verified does not hold [30]. Furthermore, model checkers and *SAT* solvers are generally available, and some finely tuned implementations are available in the open source community, including the SAT4J *SAT* Solver [158] and the *NuSMV* model checker [65]. A survey of model checkers and other tools for formal verification of real-time systems is provided in [244]. While model checkers and *SAT* solvers have similarities, their modeling and verification strategies differ significantly. Model checkers and *SAT* solvers were selected as the analysis engines for the proposed framework because they represent two classes of mature and widely used analysis engines, from two distinct communities.

2.7.1 Model Checkers

Model checkers are a class of analysis engines where the modeling formalism is a variant of finite state automata [232] and the properties to be verified are expressed using a variant of temporal logic [137]. Model checkers provide reachability analysis facilities to establish liveness and safety properties of transition systems [67]. Model checkers rely heavily on the ability to generate a finite state abstraction of the transition system model, which is then explored in a heuristic or systematic fashion [30]. The parallel combination of finite state automata gives rise to the infamous "state explosion problem", although the increase in computing power and the improved sophistication in state exploration algorithms has rendered model checkers applicable to problems of industrial size [69]. The popularity of model checkers can be attributed to the complete automation of the verification procedures and to the automated generation of a counterexample if a property of the model does not hold.

UPPAAL

The UPPAAL tool suite is a modeling and analysis environment, including model checking, for real-time systems [24, 157, 211]. Like all model checking systems, UPPAAL is composed of a modeling formalism and a temporal logic. The modeling formalism of UPPAAL is a variant of Alur-Dill automata [5]. Alur-Dill automata, also called *timed automata*, are an extension of finite state automata with real-valued clocks to express the passage of time. The timed automata of UPPAAL extends networks of Alur-Dill automata with datatypes, communication channels, and location types [157]. UPPAAL has been used as a verification engine for other formalisms such as Time Petri Nets [100]. The temporal logic [212] used in UPPAAL is a subset of Timed Computation Tree Logic (TCTL) [244], with facilities to express predicates over real-valued clock variables [29]. TCTL is the timed extension of Computation Tree Logic (CTL) [137]. The version of UPPAAL used in this thesis is version 4.0.6, released on March 5th, 2007, and available on the UPPAAL web site (http://www.uppaal.com). The UPPAAL model checker is used in the presented research to verify the timing properties of TASM models, as explained in Section 5.3.

2.7.2 *SAT* Solvers

The satisfiability problem, also known as *SAT* for short, is one of the archetypical *NP-Complete* problem in the theory of computation [232]. The problem involves determining whether a Boolean formula is satisfiable. A *Boolean formula* is composed of a set of atomic propositions and operations. Atomic *propositions* are Boolean variables that can take the values *TRUE* or *FALSE*. The propositions are connected using parentheses and the operators *NOT*, *AND*, and *OR*, represented by the symbols $\neg$, $\wedge$, and $\vee$. A Boolean formula is *satisfiable* if there is an assignment of values to propositions which makes the formula *TRUE*. If no such assignment exists, the formula is *unsatisfiable*. A sample *SAT* problem is shown below. The proposition b_i represent Boolean variables:

$$(b_1 \vee b_2) \wedge (b_1 \vee b_3)$$

The *SAT* problem has found applications in artificial intelligence and in formal verification [140]. The general interest of the *SAT* problem has led to the development of commercial and academic *SAT solvers*, which are extremely efficient analytical engines used to determine the satisfiability of Boolean formulas [175]. These solvers are heavily optimized using heuristics that can yield acceptable performance in a number of cases. The standard input format for many *SAT* solvers requires that the Boolean formula must be in conjunctive normal form (CNF). As for model checkers, *SAT* solvers rely on the fact that Boolean formulas are finite state. As opposed to model checkers, who are used to verify the properties of a state-transition model by computing a transitive closure of the system, *SAT* solvers are used to reason about sets of constraints. *SAT* solvers find a state that satisfies constraints whereas model checkers find a state of the model, reachable from initial conditions. *SAT* reasoning makes no reference to initial states or to transition rules unless they are included as constraints. *SAT* solvers have been heavily optimized and have been standardized [140]. *SAT* solvers have been used for a variety of automated analysis, including test case generation [149], [213]. Although the *SAT* problem is known to be NP-Complete, the use of *SAT* solvers has been shown to be practical in a wide range of cases.

SAT4J

The SAT4J *SAT* solver [158] is an open source solver fully implemented in Java. The solver has a well-documented API such that the solver can be easily integrated into other tools. The solver incorporates the architecture presented in [87] and has performed well in *SAT* solving competitions. The SAT4J *SAT* solver is used in the presented research, mostly because of its Java library support and because of its performance. The SAT4J solver is used to verify Completeness and Consistency [123, 125], as explained in Section 5.1, to verify resource consumption, as explained in Section 5.4, and for test case generation, as explained in Chapter 7.

SAT solvers and model checkers show similarities in their benefits, namely automation of the verification procedure and automation of the counterexample generation. *SAT* solvers and model checkers also show similarities in their drawbacks, namely the potential for state space explosion and the resulting intractability of large state space exploration.

2.8 Case Studies

The research presented in this thesis is evaluated using three case studies from three relevant domains. The case studies have been selected to reflect the typical embedded real-time systems that are targeted by the research. The first case study, the Production Cell, comes from the industrial manufacturing domains and is a problem used to evaluate formal methods in the research community. The second case study, an Electronic Throttle Controller (ETC), comes from the automotive domain and is an embedded controller used to optimize fuel consumption in automobiles. The final case study, the Timeliner Script Executor, comes from the aerospace domain and is a scripting environment in use on the International Space Station (ISS). This section provides background information about these three case studies.

2.8.1 The Production Cell

The production cell system is an industrial case study that has been used to evaluate formal methods [163]. The functional aspects of the system have been modeled and analyzed in details using Abstract State Machines (ASM) in [45]. However, the time and resource behavior have not been modeled using ASM. The system is based on an industrial metal processing plant near Karlsruhe in Germany. The production cell consists of a series of components that need to be coordinated to achieve a common goal of stamping metal blocks. Blocks come into the system as *raw* and must leave the system as *stamped.* The schematic view of the production cell system is shown in Figure 2-7. Blocks are introduced into the system via the *loader*, which puts blocks on the *feed belt.* The feed belt carries blocks from one end of the belt to the other. Once a block reaches the end of the feed belt, the *robot* can pick up the block and insert it into the *press*, where the block is stamped. Once a block has been stamped, the robot can pick up the block from the press and unload it on the *deposit belt*, at which point the stamped block is carried out of the system.

All components operate concurrently and must be synchronized to achieve the system's goals. The robot has two arms, arm a and arm b, which are perpendicular, move in tandem and can pick up and drop blocks in parallel. For example, the robot can drop a block in the press while picking up a block from the feed. To pick up or drop a block, the robot arms must extend and magnets attached to each arm must be turned on and off. A *controller* coordinates the actions of the system by using actuators to operate the various components. The original problem definition includes various safety requirements with respect to the actuators. For example, blocks must be dropped only in the press and on the deposit belt and nowhere else. The safety requirements of the original definition are listed in Section 8.1.2.

To make the TASM model easier to grasp, some simplifications and extensions have been made to the original problem definition from [163]. For example, the elevating rotatory table has been omitted. The *traveling crane* has been replaced by

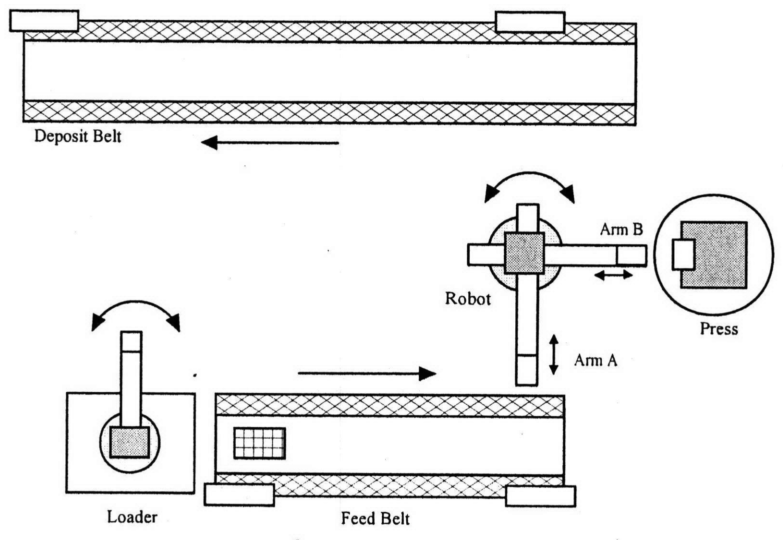


Figure 2-7: Top view of the production cell

Name	Type	Purpose
motor_press	electric motor	operate the press
motor_arma	electric motor	extend and retract arm a
motor_armb	electric motor	extend and retract arm b
magnet_arma	electromagnet	pick up and drop arm a
magnet_armb	electromagnet	pick up and drop arm b
motor_robot	electric motor	rotate robot
motor_feed	electric motor	activate and deactivate feed belt
motor_deposit	electric motor	activate and deactivate deposit belt

Table 2.2: List of actuators used in the production cell system

a *loader*, which is a component that simply puts a finite number of blocks on the feed belt. We describe every component in details in following subsections. The controller reads the state of the various components through a set of sensors and commands the various components through actuators. The set of sensors is shown in Table 2.4 and the set of actuators is shown in Table 2.2.

Electromagnet actuators can be on/off. Motor actuators can also be on/off but also have a binary direction, called *polarity*. The polarity of the motors determines the direction of the actuation. For example, setting the polarity of the *motor_arma* motor to *negative* and starting the motor will retract arm a. The combination of polarities for the motors are shown in Table 2.3.

The switch and photoelectric cell sensors are discrete binary sensors that give true/false information. The potentiometer sensors return a numerical value. The model remains faithful to the reality of sensors, actuators, and components. The controller uses only sensors and internal variables to make decisions. Furthermore, the controller uses only actuators to command the components. Sensors are read-only for the controller and actuators are read/write. Each component, other than the controller, update the values of sensors. Actuators are commanded only by the controller. This convention is congruent with the controller-environment separation principle [208].

The original example has been extended to reflect the reality that certain actions are *durative*, that is, they take a finite amount of time to complete. For example, the time that it takes for the press to stamp a block is 11 time units. The example has also been extended to include a resource, *power consumption*. For example, turning

Name	Polarity	Meaning
motor_press	positive	operate the press
motor_press	negative	operate the press
motor_arma	positive	extend arm a
motor_arma	negative	retract arm a
motor_armb	positive	extend arm b
motor_armb	negative	retract arm b
motor_robot	positive	rotate robot counterclockwise
motor_robot	negative	rotate robot clockwise
motor_feed	positive	activate feed belt in the direction loader to robot
motor_feed	negative	activate feed belt in the direction robot to loader
motor_deposit	positive	activate deposit belt in the direction out of system to robot
motor_deposit	negative	activate deposit belt in the direction robot to out of system

Table 2.3: Behavior of actuators based on polarity

Name	Type	Purpose
robot_angle	potentiometer	the position of the robot
press_status	switch	whether the press is busy or not
arma_position	potentiometer	how far has arm a extended
armb_position	potentiometer	how far has arm b extended
feed_begin	photoelectric cell	is there a block at the beginning of the feed belt
feed_end	photoelectric cell	is there a block at the extreme end of the feed belt
deposit_begin	photoelectric cell	is there a block at the beginning of the deposit belt
deposit_end	photoelectric cell	is there a block at the extreme end of the deposit belt

Table 2.4: List of sensors used in the production cell system

Component	Action	Duration	Power
Loader	Put a block on the belt	2	200
Feed	Move block	5	500
Deposit	Move block	7	500
Robot	Rotate 30°	2	1000
Robot	Extend arm	3	1200
Robot	Retract arm	2	1100
Robot	Drop a block	2	800
Robot	Pickup a block	3	1000
Press	Stamp a block	11	3000

Table 2.5: Durative actions

on the press motor consumes 1500 units of power per time unit while the press stamps a block. The list of durative actions, with their power consumptions, are shown in Table 2.5.

All other actions are assumed to be instantaneous and are assumed to consume no power. The controller actions are assumed to be instantaneous. While these assumptions do not reflect reality, it is nevertheless reasonable because the timing of the software is fast enough in relation to the timing of other components. The software operates on the order of micro seconds while the hardware components operate on the order of tenths of a second. This simplification is part of the original case study definition in [163].

Loader

The behavior of the loader is to put blocks on the feed belt. The design of the loader puts blocks on the belt either continuously or loads a specific number blocks and stops after the blocks have been loaded. The loader is used as the environmental component which *drives* the system. The behavior of the loader is to put a block on the feed belt as soon as the feed belt is empty, that is, as soon as the robot picks up a block from the feed belt. This behavior ensures that a block will be available to the robot as soon as possible so that the robot doesn't have to wait indefinitely. The loader also communicates whether or not it is done putting blocks on the belt so that the controller can take appropriate action.

Feed Belt

The feed belt is a simple component that takes a block from the loader to the robot. The feed belt is activated by the *motor_feed* motor. If the *motor_feed* motor is turned on and its polarity is *positive*, the belt moves from the feed to the robot (left to right in Figure 2-7). The belt contains two sensors, one to determine whether there is a block at the beginning of the belt (*feed_begin*) and one to determine whether there is a block at the end of the belt (*feed_end*). Some of the requirements of the belt is that it be stopped before the loader puts a block on it and that it be stopped before the robot picks up a block from it.

Robot

The robot is made up of 3 components which operate in parallel - the base, arm a, and arm b. The base can rotate clockwise and counter clockwise depending on the polarity of the *motor_robot* actuator. If the polarity of the *motor_robot* actuator is *negative*, the robot rotates in a clockwise direction in Figure 2-7. The *positive* polarity rotates the robot counter clockwise. Requirements on the robot rotation are such that it shouldn't rotate while the arms are extended in order to avoid collisions with the press and the belts.

The rotation of the robot can be controlled at the same time as the two arms. The two arms can be extended and retracted and their respective magnets can be turned on and off. The arms differ in their height such that only arm a can pick up from the feed belt and only arm b can drop blocks on the deposit belt. Furthermore, only arm a can drop a block in the press and only arm b can pick up a block from the press. These restrictions are congruent with the original problem definition [163]. The arms are operated using a motor whose polarity influences the behavior. For example, arm a can be extended by turning *motor_arma* on with its polarity set to *positive*. Arm a can be retracted by reversing the polarity. Arm b can be operated in a similar fashion. The magnets are used to pick up and drop blocks. For example, if arm a is extended at the feed belt, a block is available at the feed belt, and *magnet_arma* is turned on, the block is to be picked up by arm a and the feed belt becomes empty. Blocks remain picked up as long as the magnet is on. Once the magnet is turned off, the block is dropped. Requirements on the robot arms is that blocks should not be dropped in places other than the press and the deposit belt.

Press

The press is a simple component that is either busy stamping a block or idle. The press will begin stamping a block when the *motor_press* is turned on. The polarity of the motor does not affect the behavior of the press. The press also contains a sensor to indicate whether a block is ready or not. While the press is busy stamping a block, the block is *notfinished* and the press is *loaded.* Once the press has finished stamping the block, the block is *finished* and the press is *loaded.* There are no safety requirements on the press other than it should not be turned on when it is empty.

Deposit Belt

The deposit belt is identical to the feed belt, except that its polarity is reversed and the deposit belt is longer than the feed belt. In the model, the deposit belt is assumed to "magically" remove blocks from the system once they reach the end of the belt. The belt should not be turned on when it is empty and blocks should not be dropped on it when the belt in on. The belt can be turned on using the *motor_deposit* motor. When the motor has *negative* polarity, the deposit belt operates from the robot to out of the system (right to left in Figure 2-7). If the polarity is reversed, the belt operates in the opposite direction.

Controller

The controller uses the sensors and the actuators to operate the various components of the system. There are three core situations that the controller needs to handle. The first situation is the beginning of simulation, where there are no blocks in the system. In this situation, the robot shall wait with arm a at the feed belt until a block enters the system. Once a block enters the system, it should be picked up as soon as possible and loaded in the press. The second situation occurs when there is a block in the press. In this situation, the robot shall wait for another block to enter the system through the feed belt. The robot shall also wait for the press to be finished stamping the block. Arm a shall pick up a block at the feed and arm b shall pick up the block from the press. When this situation is met, the robot shall be rotated such that arm a will drop its block in the press and arm b will drop its block on the deposit belt. Once this has been achieved, the robot returns to the feed and the cycle resumes. The third situation occurs when there is a block in the press and the loader is no longer loading blocks into the system. When this situation occurs, the robot shall wait for arm b to pick up the block from the press and immediately unload it on the deposit belt. After this situation has occurred, the robot shall return with arm a at the feed belt and wait indefinitely until the first situation is encountered.

The controller can command all actuators in parallel and read the sensors at any point. The assumption that the controller actions are instantaneous is fair since the major concern is to ensure that the controller behavior is safe. Adding time to the controller actions would reduce the parallelism of the controller. The production cell case study is analyzed using the framework, and the results are provided in Section 8.1. The TASM model for the case study is provided in Appendix D.

2.8.2 Electronic Throttle Controller

The Electronic Throttle Controller (ETC) is a "drive-by-wire" system that is currently in use at a major automotive company. The ETC was initially modeled by Griffiths [111] as a hybrid system using Mathworks' Simulink and Stateflow [167]. The ETC is used to optimize fuel consumption based on a set of criteria, including environmental conditions such as temperature and altitude, the state of the vehicle such as engine RPM and speed, and driver inputs such as cruise control and gas pedal angle [48]. The throttle controller is a piece of software which sits between the operator and the engine and replaces the mechanical linkage between the gas pedal and the engine throttle. The software interprets driver input and operating conditions, through sensors, to decide on the desired angle of the engine throttle for optimal fuel efficiency.

The throttle angle governs how much air can enter the engine and, consequently, how much power is produced by the engine. The relationship between throttle angle and fuel consumption is intuitive. The angle of the engine throttle determines how much air can go in the cylinder, and hence controls the volume of the charge. Consequently, the throttle position governs the amount of torque produced. The fueling system is responsible for injecting an amount of fuel so that, immediately before combustion takes place, the Air-to-Fuel Ratio (AFR) is optimized. More specifically, the AFR should be stoichiometric (i.e., as close to 14.7:1 as possible for regular fuel) in order to allow for complete combustion, resulting in optimal efficiency. In order to optimize fuel efficiency, there are two main parameters to control: the angle of the throttle and the AFR as commanded by the fuel injectors. The ETC uses these two outputs to control the behavior of the engine.

Figure 2-8 shows the top level of the ETC model in Simulink, with the two key outputs – desired current (desired_current) and desired rate of fuel mass (dMfc). The angle of the throttle is controlled by the amount of current fed to the throttle servo. The desired current affects the position of the throttle and is determined based on the position of the gas pedal (as activated by the operator) and other external parameters (e.g., vehicle speed, O_2 concentration in the exhaust, engine speed, and temperature). The other controller output is the rate of fuel mass (dMfc). The dMfc value controls how much gas is sprayed in the combustion chamber. That value needs to be dynamically adjusted to maintain a stoichiometric combustion. The transfer function that characterizes the relationship between these two quantities (desired current and dMfc) is non-linear, and the model considered in this case study controls both factors independently.

The throttle controller uses modes to decide the control laws that govern the throttle actuation. For example, the throttle controller operates under different modes that have a priority ordering, depending on environmental conditions such as engine revolution, traction, cruise control settings, and driver input. The modes define the desired throttle angle, commanded through a current output from the throttle

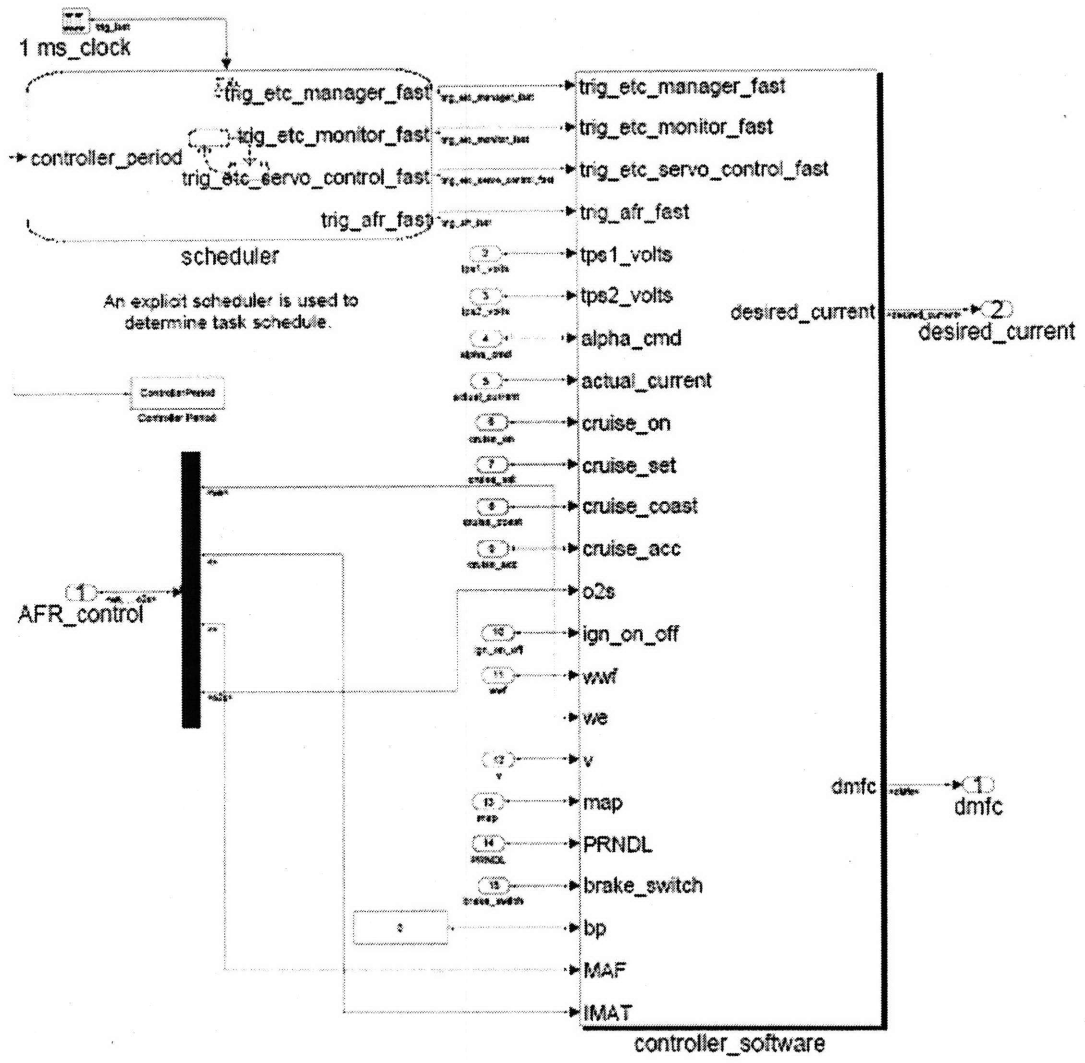


Figure 2-8: High level Simulink model of the ETC

controller.

During nominal operation, the major modes of the controller are grouped into "driving modes" and "limiting modes". The limiting modes, defined as undesirable environment conditions, take precedence over driving modes. Limiting modes include "traction control", where the wheels rotate with too little friction, and "revolution control", where the engine operates over a predefined threshold of rotations per minute. The driving modes include "human control", where the throttle is commanded via the gas pedal, and "cruise control", where the throttle is commanded based on the desired vehicle speed. The different modes are shown in Figure 2-9, adapted from [111], represented visually as a Statechart variant. The "XOR" label indicates mutual exclusion between modes and the "AND" label indicates parallel composition of modes. The transitions to the "failure detected" mode are not shown in Figure 2-9 to keep the figure simple. In each mode, a transition to the "failure detected" is possible. The detection of failure takes precedence over all other modes and the behavior of the ETC is to gradually decrease the vehicle speed until shutdown is possible.

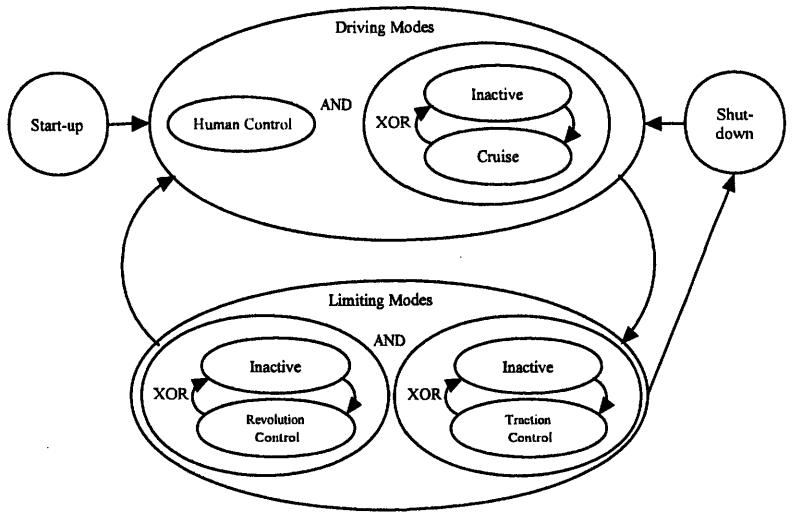


Figure 2-9: ETC modes

The modes of the throttle controller determine the desired throttle angle and, consequently, the amount of current output from the controller. The mode switching logic, as well as the calculation of the desired current represent the functional behavior of the ETC, dictating what the output should be based on various inputs. Because the calculation of the dMfc is completely isolated from the rest of the system, it is omitted from the case study. The study of the electronic throttle controller functional behavior, as well as the functional requirements for the ETC are explained in Section 8.2.

The ETC represents an interesting case study for the proposed framework because the functional behavior is implemented using a set of tasks and a scheduler. The ETC implementation is achieved using 3 tasks - a manager task, which sets the major and minor modes of the ETC, a monitor task, which periodically appraises the health of the system, and a servo task, which calculates and outputs the desired current based on the controller mode and the health of the system. The tasks have different periods and are driven by a scheduler with a 1 ms clock, as shown in Figure 2-10.

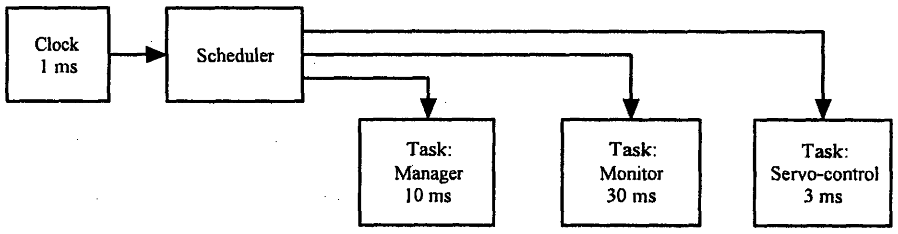


Figure 2-10: ETC tasks and scheduler

The scheduler does not support preemption and the tasks have fixed priority. The monitor task has the highest priority, followed by the monitor task, followed by the servo task. The scheduling strategy is modeled and analyzed using the framework in Section 8.3. The model in Section 8.3 contains only the scheduler and tasks and does not contain any functional behavior. Modeling the system in this way enables the verification of the scheduling strategy and the functional behavior strategy separately. In Section 8.4, the functional model and the tasking model are combined into a composite model through a series of refinements. The traceability approach described in Chapter 6 is used to demonstrate that properties that were proved separately about both the functional model and the tasking model are preserved through the combination. The three TASM models corresponding to these three views of the ETC are provided in Appendix E.

The ETC case study is used to exercise all aspects of the framework. Safety and liveness properties are verified according to the requirements expressed in [111]. Timing properties relating to the scheduling facets of the system are also studied. The model also contains 2 resources, power consumption and memory, which are analyzed for their best case and worst case conditions. Test case generation is performed based on the functional model and the tasking model. Finally, the traceability approach is exercised when combining the functional and tasking models, and is also used for regression test case generation. The ETC proves to be a interesting case study because it combines two types of models, functional and time, and utilizes all features of the framework.

2.8.3 Timeliner Script Executor

The Timeliner system [61] has been developed by the Charles Stark Draper Laboratory, in conjunction with the National Aeronautics and Space Administration (NASA), as a scripting environment to automate procedural tasks typically performed by human operators [177]. The system is composed of a high level input language, a compiler, a run-time system, and a user interface. The system is currently in use on the International Space Station (ISS) to automate a variety of tasks traditionally performed by astronauts, including spacecraft operations, subsystem checkouts, and failure detection [61].

The first component, the Timeliner language, was designed to allow easy definition of sequencing and control for complex systems. The Timeliner language is a high level scripting language with control flow based on time conditions and general Boolean conditions. Programs written in the Timeliner language are called *scripts*, and are organized hierarchically in *bundles*, *sequences*, and *statements*. Each step or decision point in a script is expressed as a series of Timeliner statements. These statements are grouped together into a Timeliner sequence, and a series of related sequences are grouped together into a Timeliner Bundle. The statement, sequence, and bundle groupings provide an organizational structure, as well as a control structure for an operator interacting with the system, as shown in Figure 2-11. A Timeliner script contains one or more bundles. A bundle contains one or more sequences and a sequence contains one or more statements. Bundles and sequences can be active or inactive. During execution, the Timeliner run-time system executes all active sequences in all active bundles. If a bundle is inactive, its sequences are not executed. In the Timeliner language, there are six general types of statements:

- Block declaration statements – these define the boundaries of bundles, sequences, and subsequences
- Timing control statements – these affect timing or flow of execution
- Conditional control statements and their modifier clauses – these allow for specific conditions that control execution based on general system values
- Action statements – these are used to carry out actions affecting the target system and support interaction with the operator
- Bundle/Sequence Control statements – these are used to manage bundles and to control sequence execution
- Non-executable statements – these are used for definitions of symbols and reserving of local storage.

A sample script is given in Listing 2.1 and other example scripts are given in Section 8.5. The complete Timeliner language is documented in [177].

On the International Space Station, the Timeliner script executor shares processor usage with other tasks. The script executor is given a fixed slice of time in which to execute sequences. One round of Timeliner execution is called a *pass*. In a pass, the script executor will sequentially execute all active sequences in all active bundles. Each sequence executes in round-robin fashion, until a blocking statement is encountered. Once a blocking statement is encountered, the execution for that sequence will resume in the next pass, at the blocking statement. Blocking statements include $EVERY$, $WHEN$, and $WHENEVER$ statements [177]. The execution times of various Timeliner statements have been heavily studied by the Charles Stark Draper

Laboratory [62]. The measures were performed using the Timeliner Testbed, with version CI_024 of the Timeliner Executor, using an embedded real-time 16MHz Intel 80836sx VME board with an 80387 floating point coprocessor. The execution times contained in document [62] are used to model the scripts in the TASM language.

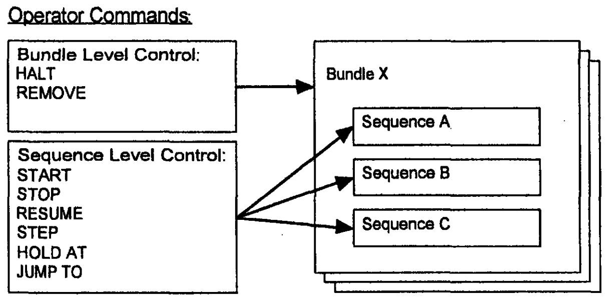


Figure 2-11: Timeliner script organization [61]

The second component, the compiler, translates an ASCII representation of the language into a form that the Timeliner execution engine can consume. The compiler additionally supports the independent definition of system data object and command information, such that Timeliner bundles can interact with a physical system without the details of the system data formats needing to be embedded within the language. Lastly, the compiler and the execution engine, known as the Executor, are designed for ease of portability to different platforms. The third component, the Executor, provides real-time monitoring and control based on the commands and conditions defined in the Timeliner sequences. A compiled bundle may be installed, executed and removed independently of execution environment software build. The Executor supports parallel execution and independent control of multiple bundles, which themselves may contain sequences that execute in parallel. This execution may be in either an asynchronous or synchronous manner. The Executor works together with the final component, the user interface, to provide the ability to precisely track, view, annotate, and interactively control an executing Timeliner script. Through the displays, an operator can also monitor the status of and receive messages from executing scripts. Hence sequences can be executed completely autonomously or via more interactive control.

Analysis

Traditionally, the Best-Case Execution Time (BCET) and Worst-Case Execution Time (WCET) of one pass of the Timeliner script execution were obtained through manual analysis and through systematic testing. However the timing properties of Timeliner scripts can be obtain through static analysis. The purpose of using the TASM language and framework is to determine the BCET and WCET for one pass of the script executor, for a given script, by taking into account the execution of all sequences and their potential interference. Determining these times will ensure that a proper time slice can be selected for the script executor. The selected time slice should be large enough to handle the worst-case scenario, but small enough to ensure optimal processor usage. To analyze the execution times of the Timeliner script executor, a set of sample scripts is modeled in the TASM language. These scripts stem from a plant controller application. The plant controller application was selected because it is simple enough to clearly explain the analysis approach but complex enough to verify interesting properties of the scripts. The details of the plant controller are detailed in the following section.

Plant Controller

The Plant Controller is a simple Timeliner application where sequences are used to maintain the cabin pressure and the ambient temperature of a plant between predefined thresholds. A logical view of the application is shown in Figure 2-12. The Timeliner script, which contains two sequences, has been obtained from [238]. The first sequence, TEMP_MONITOR, is used to maintain the temperature of the cabin between 20 and 25 Celsius degrees. The second sequence, HUMIDITY_MONITOR, is used to maintain the humidity of the cabin between 40 and 60 percent. The TEMP_MONITOR sequence is shown in Listing 2.1 and the HUMIDITY_MONITOR is shown in Listing 8.24. When the temperature is greater than 25 Celsius degrees, the sequence will command the cooling system to start. When the temperature is below 20 Celsius degrees, the sequence commands the heating system to start. The variable *TRYING_TO_COOL_SYSTEM* is used to notify the HUMIDITY_MONITOR sequence not to turn off the cooling system if the TEMP_MONITOR sequence needs it to cool the cabin. The HUMIDITY_MONITOR sequence uses the cooling system to reduce the humidity of the cabin and shares usage of the cooling system with the TEMP_MONITOR sequence.

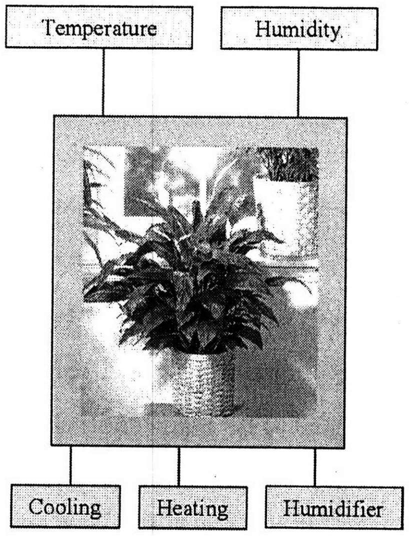


Figure 2-12: Timeliner plant application

This Timeliner script is fairly straightforward, but it is useful to demonstrate the capabilities of the TASM language and framework, notably in terms of execution time analysis. The analysis of the plant controller scripts is provided in Section 8.5. The complete TASM model of the scripts is documented in Appendix F.

2.8.4 Motivations for the Case Studies

The three case studies provide an adequate basis to evaluate the proposed framework. The case studies come from three different domains and serve to illustrate the versatility of the TASM language in modeling different applications from multiple domains. Furthermore, the combination of the three case studies provides modeling and analysis of both functional properties and non-functional properties. The production cell case study provides an application of medium size where the modeling of the hardware components remove the need to model the time behavior of the software controller. As explained in Section 2.6.1, this situation occurs because the hardware components operate on the order of seconds, while the software operates on the order of microseconds.

The ETC, on the other hand, does not model the time behavior of the environment. Consequently, the time-based behavior of the controller can be modeled and analyzed, at the task level. The ETC provides an application of industrial size that stretches the limits of the analysis capabilities, as explained in Section 8.4.7. Since the application is adapted from Mathworks' Simulink and Stateflow, it also serves to demonstrate that the TASM language can capture the semantics of Stateflow and some of the Simulink semantics. The ETC also contributes to demonstrate the modeling of scheduling and tasking alongside functional behavior.

Finally, the Timeliner case study serves to demonstrate modeling at the implementation code level, with precise timing behavior for individual code statements. In the Timeliner case study, the time-based behavior of the environment is also abstracted away and only non-deterministic changes in environment conditions are modeled. The Timeliner case study is a case study of modest size, but serves to illustrate the test case generation strategy and how it can be related to implementation code.

2.9 Segue into Chapter 3

This chapter presented background information related to the concepts used in the subsequent chapters of this thesis. Included in this chapter was information about real-time embedded systems, software engineering, and the case studies used to evaluate the presented research. Extended descriptions of the case studies, with detailed models and analysis results, are presented in Chapter 8. In the next chapter, Chapter 3, the various components of the engineering framework are presented, alongside the tool architecture used to implement the framework.

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 3

Framework Overview

This chapter presents an overview of the different components of the framework. Section 3.1 provides the motivations for the features of the framework, in light of the objectives described in Section 1.1 and in light of the target systems described in Section 2.1. Overviews of the different components of the framework are given in Section 3.3. Each component of the framework is treated in detail in subsequent chapters. This chapter also provides a description of the tool architecture that is used to implement the framework. This chapter focuses on the capabilities, motivations, and tool support of the framework while subsequent chapters describe in details the language and algorithms used to achieve the capabilities of the framework.

3.1 Introduction

The proposed framework provides a specification-based approach to system engineering by rooting engineering activities in a formal yet readable specification language. The benefits of a specification-based or model-based approach are explained in Section 2.3. Furthermore, as explained in Section 2.1, the systems targeted by the proposed framework are reactive embedded real-time systems. The aspects of interest of these systems include function, time, and resources. Consequently, the proposed framework provides the necessary facilities to model and reason about these three aspects. Moreover, the proposed framework aims to reduce the high cost of V &

V activities by integrating and automating formal verification and test case generation. The framework also addresses the inherent design paradox of model-based approaches, described in Section 2.6.1, by providing bi-directional traceability of system models from high level models down to the implementation level. The integration of a formal yet literate specification language, adequate for the target systems, formal verification, test case generation, and bi-directional traceability provide a unique set of features not available in other engineering frameworks. As explained in Section 1.1 and in Chapter 2, these features are paramount to tackle the increasing complexity and associated challenges encountered when engineering the target systems. In the following section, related frameworks are reviewed, in light of the capabilities of the framework presented in this thesis.

3.2 Related Work

The need and benefits of model-based development have prompted a variety of languages and approaches to model embedded systems, as outlined in [49, 230, 231]. In this section, related tool-supported approaches to real-time system engineering are reviewed. The comparison between competing offerings and the proposed framework is performed in the context of high level engineering capabilities. It is important to note that if a given framework does not currently provide certain capabilities, it is not necessary due to a shortcoming of the offering, but it is most likely due to the goals of the given framework and the community to which it belongs. For example, in the Ptolemy project [55], the focus is put on integrating different computation models for real-time systems and to provide an overarching simulation environment. Consequently, it is not surprising that PTOLEMY does not support code generation since it is not meant as a complete systems engineering solution. Nevertheless, popular tool-supported engineering aids, however specialized they may be, are included in this section for comparison with the proposed framework. Offerings which are not tool-supported and which are not specifically targeted at real-time systems are omitted. The offerings are compared along different dimensions, and the results are shown in Table 3.1. The specifics of each offering are compared with the offerings of the proposed framework in subsequent chapters. For example, all of the reviewed offerings are rooted in a modeling language. These modeling languages are reviewed as part of the related work in Chapter 4, where the TASM language is described. In the remainder of this section, competing approaches to the proposed framework are reviewed, in alphabetical order.

The CHARON language and associated framework and toolkit provide a rich environment to incorporate continuous and discrete dynamics for hybrid systems modeling and simulation [3]. The input language of CHARON is a variant of Statecharts [120], extended with continuous dynamics and the framework provides facilities for hierarchical modeling [6]. The CHARON suite of tools provides rich facilities for simulation of hybrid systems, including graphing facilities to visualize time-dependent behavior of continuous dynamics. The CHARON tool suite also includes a verifier, called *requiem*, which is used to explore the state space of CHARON models [4] in a model checking fashion. The primary focus of CHARON is the modeling, simulation, and verification of hybrid systems. HyTech is a symbolic model checker for linear hybrid automata [127]. HyTech's focus is primarily on modeling and verification through symbolic manipulation techniques.

The IF toolset [52] is an integrated toolset for the design and analysis of real-time systems. The IF toolset uses the Unified Modeling Language (UML) and the System Design Language (SDL) as its input specification language. The toolset translates the input languages to their own version of timed automata for analysis purposes. Recent developments have included the development of semantics for both UML and the real-time profile of UML [182] as part of the Omega project [178]. The IF toolset contains facilities for test case generation and for code generation and provides an offering similar to the framework presented in this thesis. The core differences revolve around the input languages, which are compared in Section 4.1, and in the lack of traceability and refinement concepts. Furthermore, the analysis capabilities of the IF toolset do not include execution time analysis and resource consumption analysis. Mathworks' Matlab, Simulink, and Stateflow provide a rich set of facilities for modeling and simulating hybrid systems [167]. Matlab is one of the success story of model-based engineering since it is heavily used in the embedded system industry. One of the drawbacks of Matlab is its lack of analysis capabilities beyond simple syntax verifying, type checking, and model completeness.

The PTOLEMY project and associated tool environment aims to develop a generic environment for simulation of timed systems [55]. The input language of Ptolemy can incorporate disparate computation models for the sake of hybrid system modeling and simulation. While PTOLEMY enables the integration of disparate computing models, its focus is not on system engineering. On the other hand, the Specification Tools and Requirements Methodology (SpecTRM) [160] is a suite of tools that incorporate notions of safety engineering, system engineering, and intent specification [161]. At the time of the writing of this thesis, SpecTRM contained rich facilities of system level modeling and simulation in the form of requirements, but did not yet contain facilities for software engineering activities. Consequently, the framework proposed in this thesis could represent a suitable complement to SpecTRM.

In the modeling and analysis of embedded real-time systems, the visual formalism Statecharts [120] and its associated tool STATEMATE [122] represent one of the early formalisms applied to real-time system engineering [91]. Statecharts have been heavily used in various domains and numerous semantics have been suggested and adopted in different communities [122]. STATEMATE has transitioned to industry and has become a rich tool suite for embedded systems engineering, which contains facilities for test case generation and for code generation [121]. STATEMATE does not contain refinement and traceability facilities and its input language, Statecharts, belongs to a difference class of languages than the TASM language. The two languages are compared in Section 4.1.

The Timed Input Output Automata (TIOA) is a language and mathematical framework for the modeling and analysis of hybrid systems [147]. The framework has been implemented through a set of tools using the PVS theorem prover and the UPPAAL tool suite for analysis [171]. More recently, the Tempo language and associated toolkit have been developed on top of the TIOA formalism [165]. The main focus of the TIOA framework and associated language and toolset has been on the composition semantics and proof methods to ensure correctness of TIOA models, their composition, and their refinements. TIOA's rich semantics and flexible analysis capabilities could serve as an analytical basis for integrating continuous behavior with the TASM language. By expressing TASM semantics using TIOA, hybrid systems modeling and verification could be incorporated in the framework. This option is investigated in Chapter 9 as part of future work. At the time of this writing, the TIOA toolkit and the Tempo toolset did not contain facilities for test case generation or code generation.

The UPPAAL tool suite utilizes a variant of networks of Alur-Dill automata extended with finite variables, data structures, communication channels, and urgent and committed locations [29]. The UPPAAL tool suite is comprised of an editor, a simulator, and a sophisticated verifier which explores the state space of the timed automata networks using TCTL [24]. Efforts have been undertaken to develop engineering solutions on top of UPPAAL , namely the TIMES toolset, which is a toolset to describe a scheduler and a set of tasks that can be analyzed for schedulability and synthesized to an implementation in C [9]. Furthermore, the COVER toolset is a tool to generate test cases based on networks of timed automata [128]. However, these separate offerings are not integrated into a cohesive offering and UPPAAL remains largely an analysis engine, a model checker for timed automata, with various special purpose tools developed on top of it.

3.3 Capabilities

This section provides an overview of the capabilities of the presented framework. The capabilities include modeling and simulation facilities for the target systems, static analysis of system models, bi-directional traceability of model features, and automated test case generation. Each subsequent section provides an overview of the different capabilities and provides forward pointers to following chapters describing the features in details.

Name	Hybrid Model.	Simulation	Model Checking	Other Analysis	Traceability	Refinement	Test Case Gen.	Code Gen.
TASM		x	x	x	x	x	x	
CHARON	x	x	x	x				
HyTech	x	x	x					
IF		x	x	x			x	x
Matlab/ Simulink	x	x		x				x
Ptolemy	x	x						
SpecTRM	x	x		x	x			
STATEMATE		x					x	x
TIOA/Tempo	x	x	x	x		x		
UPPAAL		x	x				x	x

Table 3.1: Comparison of the proposed framework with other frameworks for embedded real-time systems engineering

3.3.1 Modeling and Simulation

The proposed approach to real-time system engineering promotes the use of models in all phases of the engineering lifecycle. More specifically, models are used as the primary abstraction to capture desired system behavior. During different lifecycle phases, modeling occurs at different levels of abstraction, as depicted in Figure 2-4. Consequently, an appropriate modeling language should be versatile enough to express system behavior at different levels of abstraction. Furthermore, the notion of *system* is a generic notion which can include environment behavior depending on where the system boundary is drawn. As a result, an appropriate modeling language should also include facilities to define the system boundary arbitrarily, depending on the system being engineered, and to include the modeling of environment behavior as needed. In the rest of this thesis, the term *system* is used to describe the behavior captured in the specification of the system, which may or may not include a subset of the behavior of the environment.

In the proposed framework, models are expressed using the Timed Abstract State Machine (TASM) language, a novel specification language whose syntax and semantics are described in Chapter 4. The TASM language is an extension of the theory of Abstract State Machines (ASM) [42], adapted for the specification of embedded real-time systems. ASMs provide a readable specification language that can model behavior at various levels of abstraction [41], and includes a generic theory of refinement [43]. ASMs provide a flexible and generic computing model that can easily be tailored to suit a particular purpose. The TASM language extends the theory of Abstract State Machines by adapting the language to the specification of the target systems.

Because the systems targeted by the research are embedded real-time systems, functional and non-functional properties are an integral part of the system's correctness, as explained in Section 2.1. Consequently, the modeling of non-functional properties is an important feature of the proposed framework. The non-functional properties that can be expressed in the TASM language are time and resource consumption, two concepts which are added to the theory of ASMs. Because the TASM language describes behavior as the computing steps of an abstract machine, models expressed in the TASM language are executable by definition, a desirable property of system models [98]. The simulation capabilities of the proposed framework include the specification of environment behavior, encoded in the TASM language. Furthermore, scenarios depicting different initial conditions are an integral part of the simulation strategy. The details of the TASM language, alongside illustrative examples, are given in Chapter 4.

While the modeling and simulation of system behavior provides a practical and insightful approach to system engineering, using solely modeling and simulation to gain insight into system behavior can be error-prone since it relies heavily on the intellectual prowess of the user [135]. Consequently, the ability to perform analysis of system models is an important companion to simulation. The proposed framework provides a rich set of verification capabilities, as explained in the following section.

3.3.2 Static Analysis

The modeling facilities of the framework and the TASM language center around the expression of functional behavior, time, and resource consumption. The analysis capabilities include automated analysis of functional behavior in the form of completeness and consistency, two important properties of system specifications [123, 125]. These two properties are formally defined in Section 5.1 in the context of the TASM language and a verification approach is provided to automatically analyze the completeness and consistency of TASM specifications. Furthermore, because the verification approach includes model checking facilities [67], verification of functional properties using temporal logic formulas [212] is provided by the framework. More specifically, formal verification of the safety and liveness of TASM specifications, using a subset of Timed Computation Tree Logic (TCTL) [244], is achieved by reusing the UPPAAL tool suite, as explained in Section 5.2.

The framework provides a rich set of analysis capabilities for functional properties, but also addresses the analysis of time and resource consumption. The analysis of timing properties includes the automated derivation of Best Case Execution Time (BCET) and Worst Case Execution Time (WCET), using an approach called *iterative bounded liveness*, as explained in Section 5.3. The analysis of resource consumption properties includes the automated derivation of best case and worst case resource bounds for a given TASM model. Together, the analysis facilities comprise a set of algorithms and approaches to verify properties of the three key aspects of target systems, namely function, time, and resources. The complete analysis capabilities of the presented framework are explained in Chapter 5.

3.3.3 Bi-Directional Traceability

Because modeling typically happens at different levels of abstraction, ensuring consistency between different models can greatly enhance model maintenance [138]. Furthermore, the benefits of traceability between levels of abstraction has been discussed in Section 2.6.1, in terms of visualizing the propagation of design assumptions and the propagation of changes, and in terms of ensuring notions of equivalence between models. The traceability approach provided by the framework integrates notions of syntactic change with facilities to ensure semantic integrity between models. The integration of these two properties is an often overlooked problem in pure theories of refinement [43]. The bi-directional traceability capabilities provided by the framework supply a set of common refinement types that can explain the differences between two models. The two models are related syntactically by mapping the features of the models, achieving traditional notions of traceability [217]. The traceability approach can be used to track syntactic changes between models and to follow the propagation of changes and assumptions.

Furthermore, the traceability approach provided by the framework complements pure syntactic mappings with notions of semantic equivalence. For each type of refinement used to explain the differences between models, a set of correctness criteria are provided. If these correctness criteria hold for the refinement, a notion of semantic equivalence is guaranteed between the two models. The specific notions of semantic equivalence are explained in Section 6.2.2, for each type of refinement. The key idea behind the semantic equivalence approach is to reduce the verification activities. More specifically, if verification was performed on a given model, and this model is refined and traced using the proposed approach, verification results will hold in the refined model if the correctness criteria are met for the refinement. Consequently, verification performed on models before they are refined does not need to be repeated in refined models, reducing the total amount of verification that needs to be performed. In summary, the proposed approach to traceability provides bi-directional traceability so that the effects of assumptions and changes can be propagated top-down or bottom-up, verification results can be reused, and regression test cases can be generated, as explained in the following section. The traceability approach is explained in Chapter 8.

3.3.4 Test Case Generation

Simulation and static analysis of system models provide practical and insightful means to gain insight into system behavior. However, model simulation relies on the intellectual discipline of the end-user to provide all necessary scenarios to exercise the relevant system behavior. This situation can lead to important simulation scenarios being overlooked. Furthermore, while formal analysis provides mathematical guarantees that a model has certain properties, scalability remains a hurdle of automated formal analysis approaches such as model checking [69, 117, 124]. Consequently, simulation could lead to error-prone validation and formal verification might not be feasible on complex models. Nevertheless, simulation provides a lightweight and intuitive approach to validation while automated formal analysis is desirable when it can be applied. Given the limitations of simulation and of formal analysis, other means of ensuring confidence into the system are necessary as a complement. In the engineering community, the main V & V activity remains testing, in the form of unit testing, integration testing, and regression testing [242]. For safety-critical systems, testing is mandatory for certification and requires that the testing approach exercises the system to a given level of coverage [216].

In a sense, testing resembles simulation since it involves devising a scenario and observing the response of the system. However, the construction of test cases can be done systematically, to exercise the system under test to a certain degree of confidence. The presented framework provides facilities for the automated generation of test cases for unit testing, integration testing, and regression testing. The approach to automatically generate test cases uses novel algorithms that utilize TASM models to derive test cases for unit and integration testing, using the rule coverage criterion from the ASM community [103], as explained in Section 7.2. Furthermore, the framework uses the bi-directional traceability approach to identify the effects of changes at different modeling levels, so that regression testing can be automated, as explained in Section 7.6. The approach to generate test cases is described in Chapter 7.

3.4 Tool Architecture

The features of the framework are implemented in the TASM toolset [200]. The TASM toolset uses literate and graphical facilities to create, edit, simulate, and analyze TASM specifications. The toolset is comprised of multiple components, divided into front-end components, back-end components, and 3rd party analysis engines, as depicted in Figure 3-1. The TASM toolset is completely written in the Java programming language, and uses the Eclipse [183] graphical libraries for the Graphical User Interface (GUI). The TASM toolset can be used on Windows XP and Vista and on Linux. The TASM toolset is an open source project which is available, free of charge, from the TASM web site [88].

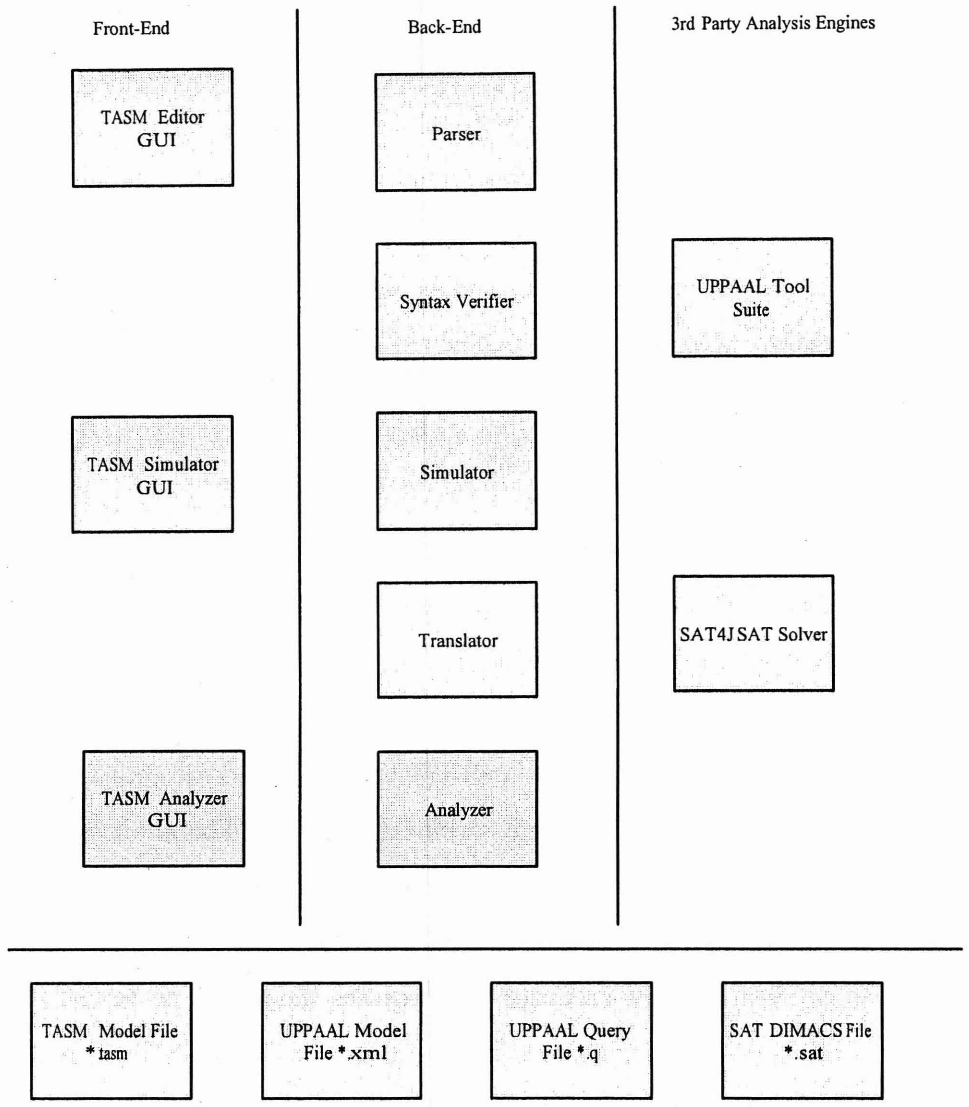


Figure 3-1: Architecture of the TASM toolset

3.4.1 Front-End Components

The front-end components of the toolset include facilities for creating and editing TASM specifications, through the TASM Editor. The editor enables the specification of functional and non-functional behavior, with standard facilities for syntax highlighting and syntax checking. The TASM Simulator enables the graphical visualization of the dynamic behavior expressed in the specification in a step-by-step fashion. Because time and resources can be specified using intervals, that is, using a lower bound and an upper bound, the simulation can use different semantics for time durations and resource consumption. For example, a given simulation can use the worst-case time (upper bound) for all steps, to visualize the system behavior under the longest running times. Other options include best-case time, average-case time, and using a time non-deterministically selected from the specified interval. The same semantics can be selected for the resource consumption behavior.

The TASM Analyzer is the component of the TASM toolset that performs analysis of specifications. The analyzer can be used to verify basic properties of TASM specifications such as consistency and completeness [123]. In the TASM language, completeness ensures that for all classes of monitored variable values, a rule will be enabled. Consistency ensures that for all classes of monitored variable values, one and only one rule is enabled. In other words, verifying consistency means verifying that the rules of a given machine are mutually exclusive. Both completeness and consistency are verified at the machine specification level. The analyzer GUI provides intuitive feedback to the user so that if a machine is incomplete or inconsistent, a witness counterexample is automatically generated from the back-end components and displayed to the user. The analyzer also provides the ability to export the completeness and consistency problems to a flat file, using the DIMACS file format, which is a standardized file format for *SAT* solver input [158].

The TASM analyzer also provides graphical facilities to verify the execution time of TASM models. The execution time is verified by mapping TASM specifications to the timed automata formalism of UPPAAL . The analyzer also provides facilities for exporting the generated UPPAAL model so that the model can be used for further analysis such as functional verification using temporal logic formulas. The derivation of minimum and maximum resource consumption is also provided by the TASM analyzer through a GUI.

3.4.2 Back-End Components

The back-end components of the TASM toolset provide most of the facilities available in the Graphical User Interface (GUI). The parser is responsible for loading and saving the TASM model to disk, using the "*.tasm" file format whose context-free grammar is available in Appendix A. The syntax verifier is used to ensure that errors in the models can be easily identified in the TASM editor through syntax highlighting and detailed error messages. Once the syntax has been verified to be free of errors, the simulator can explore various behaviors of the TASM model, through step-by-step analysis and different initial conditions specified through the TASM simulator. The simulator provides a rich interface, including the list of generated update sets, the history of resource consumption, and the values of internal and external state components.

The analyzer is the bridge between the GUI and the 3rd party engines. The analyzer provides all of the necessary facilities so that a user of the toolset is unaware that 3rd party engines are used in the analysis procedure. The analyzer provides a rich interface to the TASM analyzer GUI so that feedback can be provided to the toolset user in an intuitive fashion. The back-end analyzer achieves its tasks by translating TASM models through the back-end translator. The translator is the back-end component used to map between the TASM syntax to the syntax of 3rd party engines. The translator understands the 4 main file formats used in the TASM toolset, namely the TASM file format (*.tasm), the DIMACS file format (*.sat), the UPPAAL model file format (*.xml), and the UPPAAL query file format (*.q). The DIMACS file format is the standard input format of *SAT* solvers, including the SAT4J *SAT* solver. The UPPAAL tool suite uses its own version of XML as its input format and saves temporal logic formulas in a separate file, called the query file (*.q). The translator juggles these different formats to provide the necessary facilities to the back-end analyzer and to provide import/export capabilities to the analyzer front-end.

3.4.3 3rd Party Engines

The analysis of completeness and consistency is achieved by translating machine rule guard expressions into a Boolean formula in conjunctive normal form [192]. The Boolean formula can then be verified for satisfiability using a *SAT* solver. The TASM toolset uses the SAT4J solver, an open source *SAT* solver [158]. The completeness and consistency problem is formulated in such a way that an incomplete or inconsistent specification leads to a satisfiable Boolean formula. Formulating the problem this way ensures that the *SAT* solver can automatically generate a counterexample if the specification is inconsistent or incomplete. The SAT4J solver is a Java-based solver which can be integrated seamlessly into any Java application. The TASM toolset provides the option to solve the completeness and consistency problems directly, without requiring the user to know that the specification is being translated to *SAT* . Because the input format of *SAT* solvers is standardized, the TASM toolset provides the capability to export the generated *SAT* problem, so that the problem can be analyzed and solved outside of the toolset. The mapping SAT4J *SAT* solver is also used to obtain minimum and maximum resource consumption.

The analysis of execution time is achieved with the UPPAAL tool suite. The UPPAAL tool suite is also written in Java, but is not an open source project. Furthermore, the UPPAAL tool suite does not have a public interface that can be used to manipulate UPPAAL models programmatically. However, UPPAAL contains an official library to manipulate the XML model file. The UPPAAL verifier is the component of UPPAAL used to explore the state space of the timed automata model and contains a published interface to load connect, load models, and execute temporal logic queries against the model.

3.5 Segue into Chapter 4

In this chapter, an overview of the framework was presented, including high level descriptions of each component of the framework. Furthermore, the tool architecture used to realize the framework was presented. Each component of the framework is treated in-depth in the following chapters. The next chapter, Chapter 4, describes the language that is used as the specification basis for the framework, the Timed Abstract State Machine (TASM) language.

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 4

The Timed Abstract State Machine Specification Language

This chapter describes the Timed Abstract State Machine (TASM) language, the modeling language which serves as the specification basis for the presented framework. The TASM language an extension of Abstract State Machines (ASM) to include facilities for specifying time and resource consumption. This chapter presents the motivations for the choices of the concepts included in the language, in light of target systems and related work. The syntax and semantics of the TASM language are presented, accompanied by an illustrative example to explain the concepts as they are introduced. The descriptive example concerns the behavior of a light and fan controller and is detailed in Section 4.1.5.

4.1 Related Work

This section presents a large body of related work concerning the design of the TASM language. The following subsection presents a review of usability concepts for specification and modeling languages, as support for the usability potential of the TASM language. Since the TASM language is an extension of ASM, a brief overview of ASM is presented, with related work concerning the inclusion of time in the ASM formalism. An overview of the main features of the TASM language are provided, in order to qualify the differences between TASM, ASM, and related formalisms. The overview serves as an introduction to the motivations and features of the TASM language before the language is explained in details in Section 4.2 and in Section 4.3. Finally, this section concludes with a comparison of TASM with other popular formalisms used for the modeling and analysis of embedded real-time systems, for the frameworks presented in Section 3.2.

4.1.1 Usability of Specification Languages

The term "Formal Methods" has historically been used to designate an approach to system specification based on rigorous mathematical principles with an associated proof system to mathematically reason about properties of the system under design [225, 248]. The benefits of formal methods have been heavily documented, including specifications which are unambiguous and the ability to uncover defects during the early phases of the engineering lifecycle [50, 117]. However, the mathematical nature of early efforts in the formal methods community have yielded a set of languages and proof systems that were challenging to use by practitioners not versed in mathematics or computer science [71, 124]. But the benefits of unambiguous specifications, combined with the ability to detect defects early in the engineering cycle have provided a value proposition attractive to practitioners. Consequently, efforts have been made to combine the rigor of formal methods with a specification language that can be readily used by practitioners [162].

The topic of "usability" of a specification language is a highly subjective subject and depends heavily on the experience of the specifier and on the quality of the tool supporting the specification activities. Nevertheless, some basic notions of readability of specification languages have been established in the literature [146]. The term *literate* is used to denote a specification that can be *read*, like the English language [151]. Textual languages are traditionally more readable than graphical notations such as statecharts, which can become cluttered, counterintuitive, and have no clear starting point and end point for reading purposes [258]. The TASM language was designed with readability in mind by avoiding the use of special symbols, keeping the syntax minimal, and providing abstraction mechanisms to structure specifications [124, 146, 240]. Furthermore, in terms of real-time system specification, the language makes the expression of timing concepts explicit, a desirable property of real-time languages [254]. While no controlled experiments were conducted to investigate the usability of the TASM language, the principles of literate specifications were maintained during the design of the language. Furthermore, since the TASM language is based on ASM, the usability of TASM can be inferred from the past successes of ASM in terms of readability [41]. Experience with industrial contacts, with fellow undergraduate and graduate students, and with presentations of the TASM language in various communities have served to reinforce the assumption that the TASM language is a literate specification language which can be used and understood by almost anyone who has basic programming proficiency [42].

4.1.2 Abstract State Machines

Abstract State Machines (ASM) provide an approach to specify, analyze, and verify hardware and software systems at different levels of abstraction [42]. The motivations and benefits of using Abstract State Machines (ASM), formerly known as evolving algebras, for hardware and software design have been documented in [41]. On the practical-side, ASMs have been used successfully on a wide range of applications, ranging from hardware-software systems to high level system design [42, 47]. ASMs have also been shown to be scalable to industrial size systems [46]. Furthermore, there is enough evidence to believe that ASMs provide a *literate* specification language, that is, a language that is understandable and usable without extensive mathematical training [71], as explained in Section 4.1.1. The preliminary evidence supporting the ease-of-use of ASMs revolves around the small size of the syntax, the simplicity of the semantics, and the avoidance of extraneous mathematical symbols. Moreover, the *semantic distance*, that is, the amount of effort required to translate between one model to another, for example between a design specification and an implementation, appears to be "small" for ASMs. The term "small" is used in comparison to other formalisms that are predominantly visual (e.g., timed automata) [258] or that make heavy use of mathematical symbols (e.g., process algebra).

On the theoretical side, ASMs have well-defined formal semantics, which makes ASM specifications unambiguous and subjectable to formal analysis. ASM specifications are also independent of a specific verification method and can be verified either through manual proofs or through automated tools [249]. Furthermore, ASM theory was developed as a methodology for high level system design [42]. Consequently, *refinement*, or the process of gradually adding details to a system design, is an integral part of the theory, which makes ASMs applicable at various levels of abstraction. Finally, ASM specifications are executable, a useful property in the construction and validation of specifications [98]. The anecdotal evidence supporting the success of the ASM method [41] suggests that tailoring the formalism to the area of embedded real-time systems could achieve similar benefits. Abstract State Machines have also been used to automate engineering activities, including verification using model checkers [249] and test case generation [110].

The TASM language is an extension of Abstract State Machines (ASM), with facilities to specify time and resource consumption. The subset of ASM included in the TASM language is the same as explained in [249], which includes conditional statements and assignments, but excludes the *forall* construct and the *choose* construct. The *forall* statement is excluded because the duration of this construct depends on dynamic conditions and cannot be statically assigned. The *choose* construct is omitted for similar reasons because it is counterintuitive to assign a static duration to non-deterministic choice. The TASM language also excludes the *import* construct because safety-critical real-time systems discourage dynamic allocation. The omission of these three constructs is not too restrictive since many ASM specifications have not used these constructs, e.g., the production cell system in [45]. The concepts of Abstract State Machines (ASM) revolve around the concepts of an abstract machine and an abstract state. For an ASM, behavior is specified as the computing steps of the abstract machine and its effects on the abstract state. More specifically, the dynamic behavior is expressed through the machine executing a *step*, which corresponds to a group of atomic updates made to global state. An *update set* is the term used to describe the set of atomic updates that are associated with a single step. A *run* of an ASM, is a sequence of steps, that is, a sequence of update sets. The global state after each step can be obtained by applying individual update sets sequentially.

The syntactical structure of a machine in the TASM language is an ASM in *canonical form*, also called an ASM in *block form* [110]. In this form, a machine is structured into a finite set of rules, written in precondition-effect style. Conceptually, block form is convenient for structuring specifications and analysis but it is not necessary since any ASM can be converted to block form by introducing a program counter variable [110]. For an ASM that contains n rules, a machine in block form has the following structure:

$$\begin{aligned} R_1 &\equiv if\ G_1\ then\ E_1 \\ R_2 &\equiv if\ G_2\ then\ E_2 \\ &\vdots \\ R_n &\equiv if\ G_n\ then\ E_n \end{aligned} \tag{4.1}$$

The guard G_i is the condition that needs to be enabled for the effect of the rule, E_i, to be applied to the environment. The effect of the rule is grouped into an *update set*, which is applied atomically to the environment at each computation step of the machine. In the ASM community, ASMs have been used to model specific examples of real-time systems [44, 72]. Some extensions have been proposed to the ASM theory to include timing characteristics [221] but the extensions make no mention of how time is to be specified (only the theoretical semantics are proposed) and do not address concurrency. Related work from the ASM community concerning using ASM to specify and analyze embedded real-time systems is reviewed in the following section.

Time in Abstract State Machines

The proposed approach to incorporate time in the Abstract State Machine (ASM) formalism incorporates concepts from a variety of previous approaches from the ASM community. In the ASM community, related work has revolved around two main paradigms: instantaneous actions with time constraints, also called *timed ASMs* [72], and durative actions [44]. In timed constrained ASMs, all actions are instantaneous but rule guards can contain predicates over an external function called *currtime*, which denotes a wall clock. The *currtime* function is a monotone function which takes no argument and returns a value from the Reals domain. This approach has been used to specify and analyze real-time concurrent algorithms such as the railway crossing problem [22] and the Kermit protocol [136]. This approach is well-suited for declarative specification and for event-based systems where the temporal duration between events is the primary representation of timed-based behavior. However, the systems targeted by the TASM language are naturally specified using a duration paradigm. The approach presented in this thesis also contains a function analogous to the *currtime* function, called *now*, but the function is not an external function, motivating the use of a different name. The underlying semantics of the *currtime* function are highly dependent on the moves of agents being durative since time progression is determined through agent actions. In timed ASMs and related approaches, the concept of time is an external function that is not part of the system behavior [114]. The progression of time is dependent on the rule guards and not on the actions of the specified system.

In contrast, the TASM language provides facilities to specify the duration of actions performed by the specified system. A similar approach using durative actions has been used in [44] to analyze Lamport's bakery algorithm. In this approach, an untimed version of the algorithm is presented and is refined with durative actions. The refinement is shown to preserve the correctness of the untimed version. The approach is based on asynchronous ASM and the notion of partially ordered runs [115]. The durative moves are specified to occur during an open real interval *(a, b)* where *a* and *b* are time values on the global time axis. Using the time specification, the moves of agents are ordered linearly and the requirements of partially ordered runs are extended to include conditions for overlapping moves. The approach presented in [44] provides no structured syntax to capture the duration of actions and the analysis of the specification relies on creative proof methods. Furthermore, the moves of agents are specified on the global time axis instead of in terms of relative duration of moves, as used in TASM.

The approach adopted in the TASM language follows a durative action paradigm but specifies moves of agents in terms of relative durations of moves. The duration of a run is thus related to the summation of the moves of agents. Furthermore, the concurrency semantics in the TASM approach is related to synchronous multi-agent ASMs [47] since the moves of agents are synchronized using a global system clock. In the TASM language, there are no external functions that are not controlled by an agent of the specification. External functions are included into the behavior of agents that represent the environment. While the lack of external functions might seem counterintuitive to model embedded controllers, the external functions have been replaced by functions controlled by agents representing the environment. In this way, the system can be simulated completely without the need to hardcode the values of external functions since the values in the environment can depend on the behavior of the system. The TASM approach resembles the *real-time controller ASM* approach where runs are extended with state changes that occur at *computationally significant real-time moments* [72]. However, the computation of the significant real-time moments is a result of the actions of agents and is not determined a priori, as is the case in [72].

The key difference between the Timed Abstract State Machine (TASM) language and ASM is that steps are *durative* in TASM. In ASM, machine steps are instantaneous. Furthermore, in TASM, durative steps can consume a finite amount of resources. In the case of single agent specifications, the durative steps of the agent dictate the progression of time in the specification. In the case of multi-agent specifications, the durative steps are used to synchronize agents with respect to one another. In TASM, a step is the execution of a rule, which produces an update set. The update set is applied atomically to global state. For the single agent case, the duration of the step, reflected in the update set obtained through a rule execution dictates the progression of time. At the completion of a step, the environment is updated by applying the update set once the step duration has elapsed.

The concept of step is fundamental in the definition of ASM and in computation theory in general since a step defines the atomic unit of progression of an abstract machine. In TASM, the concept of step is augmented with a duration and a set of resources consumed during the step execution to capture the physical reality of embedded real-time systems. This abstract model adequately captures the physical reality of computer systems where steps are typically rarely instantaneous. The durations and resource consumptions can be easily modified to capture behavior at different levels of abstraction, to document system assumptions, and to relate models at different levels of abstraction, including non-atomic refinement. In concrete computer systems, the notion of step varies depending on the level of abstraction. For example, a step could be considered a clock cycle, a machine operation, or a statement execution in a high level programming language. Throughout this chapter, the terms *step*, *rule execution*, *move of an agent*, and *action of an agent* are used interchangeably.

The composition extensions for ASMs presented in this chapter are based on the XASM language [10]. However, the XASM language does not include time or resource specification and only deals with single agent ASMs. The specification of resource consumption has not been addressed in the ASM community.

The systems that are targeted by the TASM language are embedded real-time systems. These systems include embedded controllers that monitor the environment periodically, through sensors, and take action on the environment through actuators. The important characteristics of such systems is that the values of sensors as read by the system are directly related to the actions taken by the system. Consequently, the behavior of the environment, typically represented as *external* functions in previous ASM approaches [72], cannot be hardcoded a priori since they depend on the actions of the controller. More information about target systems is available in Section 2.1.

4.1.3 The TASM Language

At a high level, the concrete syntax of the TASM language extends the block form of equation 4.1 to include time and resource consumption. The specification of time and resource consumption is achieved through annotations of individual rules. The concrete syntax of TASM resembles the ASM syntax presented in [104], with extensions for time and resource annotations. To illustrate these concepts, a sample rule of a block TASM is shown in Listing 4.1, expressed in the concrete syntax of the TASM language. The rule describes the behavior of the feed belt from the production cell case study [163]. For a description of the production cell system and a graphical representation of its layout, the reader is referred to Section 2.8.1. The feed belt carries blocks from the loader to the robot. According to the description of the system, moving a block from the loader to the robot takes 5 time units and consumes 500 units of power. Listing 4.1 shows the rule with the time and resource annotations. The line numbers are not part of the specification and are added to ease the description of the listing.

Listing 4.1 Rule 1 of machine Feed

```
1:  R1: Block goes to end of belt
2:  {
3:    t     := 5;
4:    power := 500;
5:
6:    if feed_belt = loaded and feed_begin = True and
7:      motor_feed = on and motor_feed_p = positive then
8:        feed_begin  := False;
9:        feed_end    := True;
10: }
```

In Listing 4.1, line 1 contains the name of the rule, line 3 contains the time annotation, line 4 contain a resource annotation, line 6 and 7 contain the guard G of the rule, and lines 8 and 9 contain the effect expression E of the rule. Semantically, rule R_1 will be *enabled* when the guarding condition evaluates to *True* and when the machine is not busy executing a rule. When rule R_1 is executed, the machine will be blocked from executing other rules for 5 time units, at which point the effect of executing the rule will be applied to the environment. Furthermore, during the 5 time units of the rule execution, 500 units of power will be consumed. While a machine is "busy" executing a rule, other parallel machines, if present, can execute rules in overlapping time intervals and the durations of their rule executions determine the synchronization of parallel update sets. The semantics of rule execution are such that a rule is executed based on the state at the beginning of the rule execution. These semantics are congruent with the target systems described in Section 2.1, which "cache" the state read through sensors before making a decision about the output. Furthermore, the TASM language uses relative duration at its time specification paradigm, in the form of rule execution times. While the semantics of the TASM language could be expressed using timed constrained ASM [114], the TASM language provides a concise and readable notation to express the desired behavior of the target system. A mapping between the TASM language and ASM is provided in Section 4.4. The syntax of the TASM language is explained in detail in Section 4.2 and the semantics are explained in Section 4.3.

4.1.4 Other Specification Formalisms

In the academic community, numerous mathematical formalisms have been proposed to specify and analyze real-time systems. The most popular formalisms developed in academia can be classified into three main families: automata, process algebra, and Petri nets [33]. These three families are reviewed and the languages of the related frameworks described in Section 3.2 are compared to the TASM language, with comparison results presented in Table 4.1.

In the automata family, timed automata are finite state automata extended with real-valued clocks and communication channels [5]. The formalism has been used on a variety of applications and is the formalism used in the UPPAAL tool suite [157]. The formalism is well-suited for analysis by model-checking, but the lack of structuring mechanisms makes abstraction and encapsulation difficult to achieve [34]. Statecharts and the associated tool STATEMATE [122] augment automata with structuring mechanisms (superstates). Statecharts also include time concepts through the use of delays and timers. Statecharts have been heavily studied in various communities and many different semantics exist to describe the behavior of statechart models [23, 122].

In the Petri net family, a large number of variations on the traditional Petri net model have been developed, including various models of time [60]. Non-determinism is an essential part of Petri nets, which makes Petri net unsuitable for the specification of safety-critical real-time systems where predictability is of highest importance [34].

In the process algebra family, various offsprings of Communicating Sequential Processes (CSP) [31] and the Calculus of Communicating Systems (CCS) [170] have been defined, including multiple versions of timed process algebra [31]. However, in this formalism, it is difficult to express non-functional properties other than time (e.g., resource consumption). Timed LOTOS (ET-LOTOS) [31] is an example of a language from the process algebra family. Other well known formalisms include the Synchronous languages ESTEREL and LUSTRE [34].

In the industrial community, especially in the aerospace and automotive industries, the Unified Modeling Language (UML) [181] and the Architecture Analysis and Design Language (AADL) [223] have come to dominate notational conventions. At its onset, UML did not have formal semantics and remained a graphical language with limited support for automated analysis. Since its inception, many tools have defined their own semantics for UML, but the international standard [181] still does not contain a standard definition of the formal semantics. In the UML community, two real-time profiles have been proposed, the UML profile for "Schedulability, Performance, and Time Specification (SPT)" [180] and the UML profile for "Modeling and Analysis of Real-Time and Embedded Systems (MARTES)" [182]. The MARTES profile is the latest profile that corresponds to version 2.0 of UML. While both profiles contain a large amount of syntax, the lack of a consistent semantics, as well as disagreements among community leaders create challenges for widespread adoption of the profiles [93, 107]. Furthermore, UML is predominantly tied to object-oriented approaches [105, 108]. AADL contains formal semantics but is still in the early development stage. It is unclear whether AADL can be used to specify low level functional behavior. In its current form, AADL remains an Architecture Description Language

(ADL) and cannot express component-level behavior.

When comparing specification languages, numerous dimensions can be utilized for the comparison, including usability [162], composition models, communication model [144], and whether a language is graphical or textual. The comparison of related languages to the TASM language is performed using the categories in the headings of Table 4.1. These categories were selected based more on usability issues and less on semantic richness [162]. For example, it was argued in Section 2.1 that time specification using a duration paradigm is well-suited for the specification of the desired behavior of the systems of interest. Furthermore, a textual representation obeys the principles of literate specifications [146] while hierarchical composition is paramount to structure specifications and for reuse [35]. One of the main differences between the TASM language and languages like CHARON [4], TIOA/Tempo [147], and Simulink and Stateflow [167] is that TASM does not currently have facilities for specifying continuous behavior such as dynamics described by a differential equation. This difference is debatable since the behavior of a software system is inherently discrete. Continuous dynamics need to be included only when considering issues of performance such as stability and steady-state error [218]. Nevertheless, verification engines and notations that do not include continuous dynamics have proved useful in specifying and analyzing systems, such examples include UPPAAL [24] and UML [179]. In the frameworks described in Section 3.2 the language SDL is not included in Table 4.1 because that language is primarily applied to telecommunication protocols [172], a type of system not targeted by the TASM language. Furthermore, UML uses a variant of statecharts, as does STATEMATE [121].

4.1.5 Light Switch Example

Throughout this chapter, a small example is presented to illustrate the features of the TASM language. The example contains a light bulb, a fan, two switches and two abstract state machines that operate in parallel and control the status of the light and fans depending on the state of the switches. A schematic view of the application is shown in Figure 4-1. This example is used throughout this chapter to

Name	Continuous Dynamics	Hierarchical Composition	Parallel Composition	Representation	Time Approach	Communication Model
TASM		x	x	Textual	Duration	Shared Vars
Simulink/ Stateflow	x	x	x	Graphical	Timers	Channels
Statecharts		x	x	Graphical	Timers	Channels
TIOA/Tempo	x		x	Textual	Diff. Eq.	I/O
Timed Automata			x	Graphical	State	Channels
SpecTRM-RL	x	x		Text, Tabular	Timers	I/O

Table 4.1: Comparison of the features of the TASM language with other languages for embedded real-time system specification

illustrate concepts as they are introduced. Different versions of the example are used to illustrate different concepts. For example, version 1 of the example, in Listing 4.2, contains only the control for the light bulb, one switch, and the light bulb (the fan components are omitted). The presented example also contains two resources, *memory* and *power*, that the machines can use to perform their functions. While the presented example is quite simple, it is useful to illustrate the concepts of the TASM language. More substantial examples of TASM models are available in Chapter 8.

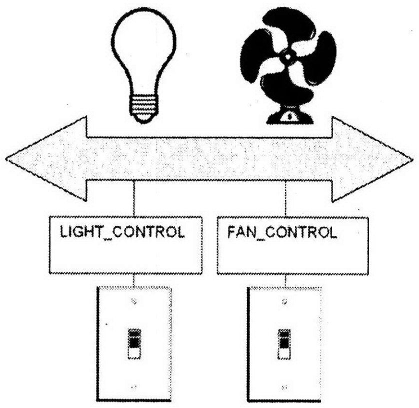


Figure 4-1: Light switch example

4.2 The Timed Abstract State Machine (TASM) Language: Syntax

This section describes the syntax of the TASM language. In Section 4.2.1, the syntax of the ASM formalism is expressed with discrete mathematics concepts, in so-called *abstract syntax*. The sample specification given in Listing 4.2 is expressed *concrete syntax* of the ASM language, that is, the syntax that can be implemented in a toolset and input via a keyboard. In this section, the same convention is followed – the TASM concepts are introduced using abstract syntax and illustrated in examples using concrete syntax. The syntax used in Listing 4.2 and used in subsequent listings is the syntax implemented in the TASM toolset. A complete description of the concrete syntax of the TASM language is available in Appendix A.

4.2.1 Basic ASM Specification

The term *specification* is used to denote the document that results from the process of writing down a system design. The term specification is used interchangeably with the term *model* throughout this chapter. This section introduces specifications that contain only a single abstract state machine, also known as *basic* or *single-agent* ASMs in the ASM community [47]. This section provides the basis for expressing the syntax and semantics of the TASM language by providing a simple definition of a specification. The specification described is equally applicable to ASM or to TASM because it does not utilize any of the features that distinguish TASM from ASM. Consequently, the material presented in this section can be interpreted as a formulation of ASM, in terms that will be useful to describe the features of the TASM language.

A basic abstract state machine specification is made up of two parts - an abstract state machine and an environment. The machine executes steps based on values in the environment and modifies values in the environment. The environment consists of two parts - the set of environment variables and the universe of types that variables can have. In the TASM language all variables are strongly typed. The machine consists of three parts - a set of monitored variables, a set of controlled variables, and a set of rules. The *monitored* variables are the variables in the environment that affect the machine execution. The *controlled* variables are the variables in the environment that the machine affects. The set of *rules* are named predicates, written in precondition-effect style, that express the state evolution logic. Formally, a specification $ASMSPEC$ is a pair:

$$ASMSPEC = \langle E, ASM \rangle$$

Where:

- E is the environment, which is a pair:

$$E = \langle EV, TU \rangle$$

  Where:

  - EV denotes the *Environment Variables*, a set of typed variables
  - TU is the *Type Universe*, a set of types that includes:
    * Reals: $RVU = \mathbb{R}$
    * Integers: $NVU = \{\ldots, -1, 0, 1, \ldots\}$
    * Boolean constants: $BVU = \{True, False\}$
    * User-defined types: $UDVU$

- ASM is the machine, which is a triple:

$$ASM = \langle MV, CV, R \rangle$$

  Where:

  - MV is the set of *Monitored Variables* $= \{mv \mid mv \in EV$ and mv is read-only in $R\}$

- CV is the set of *Controlled Variables* = $\{cv \mid cv \in EV$ and cv is read-write in $R\}$
- R is the set of *Rules* = $\{(n, r) \mid n$ is a name and r is a rule of the form *if C then A* where C is an expression that evaluates to an element in BVU and A is an action$\}$

An action A is a sequence of one or more updates to environment variables, also called an *effect expression*, of the form $v := vu$ where $v \in CV$ and vu is an expression that evaluates to an element in the type of v.

Updates to environment variables are organized in *steps*, where each step corresponds to a *rule execution*. In the rest of this chapter, the terms *step execution* and *rule execution* are used interchangeably. A rule is *enabled* if its guarding condition, C, evaluates to the Boolean value $True$. The *update set* for the i^{th} step, denoted U_i, is defined as the collection of all updates to controlled variables for the step. An update set U_i will contain 0 or more pairs *(cv, v)* of assignments of values to controlled variables.

An update set is said to be *consistent* if there are no conflicting updates in the set, that is, no variable is updated twice with different values. That is, an update set U is consistent if:

- For all two update pairs *(cv1, v1)*, *(cv2, v2)* in *U*:
  - if *cv1* = *cv2* then *v1* = *v2*

A *run* of a basic ASM is defined by a sequence, potentially infinite, of update sets. For an ASM that terminates after n steps, a run would yield a sequence of update sets at each step:

$$U_1,\ U_2,\ \ldots,\ U_n$$

The state progression can be obtained by applying the update set at each step. For an ASM that terminates after n steps, the state progression the run of the ASM yields n states:

$$S_0,\ S_1,\ S_2,\ \ldots,\ S_n$$

The state S_0 denotes the initial values of the environment at the beginning of the machine execution. The operator $\circ$ is introduced to denote the application of an update set to the current state to yield a successor state. More specifically:

$$S_i = S_{i-1} \circ U_i \qquad (i > 0)$$

The complete reference about ASM theory is available in [47].

4.2.2 Light Switch Example Version 1

Version 1 of the example contains only the light bulb and the corresponding switch. Listing 4.2 shows a basic ASM specification describing the logic for switching the light "on" or "off" based on whether the switch is "up" or "down". The specification is divided into sections, identified by capital letters followed by a colon. Comments in the specification are preceded by the escape sequence "//".

A sample run with the initial environment $((light,\ OFF),\ (switch,\ UP))$ yields one update set:

$$U_1 = ((light,\ ON))$$

The run of the machine becomes:

- $S_0 = ((light,\ OFF),\ (switch,\ UP))$
- $U_1 = ((light,\ ON))$
- $S_1 = S_0 \circ U_1 = ((light,\ OFF),\ (switch,\ UP)) \circ ((light,\ ON)) = ((light,\ ON),\ (switch,\ UP))$

After the step has finished executing, the environment becomes: $((light,\ ON),\ (switch,\ UP))$. At this point, since the machine no longer has enabled rules, the machine terminates.

Listing 4.2 Light switch example version 1

```
ENVIRONMENT:

USER-DEFINED TYPES:
  light_status  := {ON, OFF};
  switch_status := {UP, DOWN};

VARIABLES:
  light_status  light   := OFF;
  switch_status switch  := DOWN;

-------------------------------------------

MAIN MACHINE:

MONITORED VARIABLES:
  switch;

CONTROLLED VARIABLES:
  light;

RULES:

  R1: Turn On
  {
    if light = OFF and switch = UP then
      light := ON;
  }

  R2: Turn Off
  {
    if light = ON and switch = DOWN then
      light := OFF;
  }
```

4.2.3 Time

The TASM approach to time specification is to specify the duration of a rule execution. In the TASM world, this means that each step will last a finite amount of time before an update set is applied atomically to the environment. Syntactically, time gets specified for each rule in the form of an annotation. The annotation is specified as an interval $[t_{min}, t_{max}]$. The lack of a time annotation for a rule is assumed to mean $t = 0$, an instantaneous rule execution. Semantically, a time annotation is interpreted as a closed interval over $\mathbb{R}_{\geq 0}$. For a given run, a rule execution will last an amount t_i where t_i is taken non-deterministically from the interval $[t_{min}, t_{max}]$. The approach uses relative time between steps since each step will have a finite duration. The total time for a run of a given machine is simply the summation of the individual step times over the run.

Because time is used as a synchronization mechanism and because the specification denotes the behavior of reactive systems, a special keyword can be used in time annotations. This keyword, called *next*, is used with time specification, e.g. "`t := next`", to denote that the duration of a rule execution will be determined by the application of an update set generated by a parallel entity. For example, when a machine executes a rule with the `t := next` annotation, the update set produced by the rule will be applied at the time of the next state change, dictated by the update set of another machine. This time annotation can be used to synchronize parallel entities who are waiting for a handshake. Furthermore, this special time annotation can be used to denote that a given machine will wait for a state change before executing a rule. This construct could be used to specify that the machine will not do any "useful" work until some outside party alters the value of one of its monitored variables. The *next* construct essentially states that time should elapse until an event of interest occurs and is used to keep the machine "live" and prevent termination or infinite loops.

4.2.4 Resources

The specification of non-functional properties includes timing characteristics as well as resource consumption properties. A *resource* is defined as a global quantity that has a finite size. Power, memory, and communication bandwidth are examples of resources. Resources are used by the machine when the machine executes a rule. Similarly to time specification, syntactically, each rule specifies, as an annotation, how much of a given resource it consumes. The annotation is specified as an interval $[rr_{min}, rr_{max}]$. The omission of a resource consumption annotation is assumed to mean zero resource consumption. Semantically, a resource annotation is interpreted as a closed interval over $\mathbb{R}_{\geq 0}$. For a run, for each resource, a rule execution will consume an amount rr_i where rr_i is taken non-deterministically from the interval $[rr_{min}, rr_{max}]$. The semantics of resource usage are assumed to be *volatile*, that is, usage lasts only through the step duration. For example, if a rule consumes 128 kilobytes of memory, the total memory usage will be increased by 128 kilobytes during the step duration and will be decreased by 128 kilobytes after the update set has been applied to the environment. Time elapses and resources are consumed only when a rule is executed. Determining whether a given rule is activated is instantaneous and consumes no resources.

Formally, a rule R of a machine ASM, described in Section 4.2.1, is extended to reflect time and resource annotations:

$$R = \langle n, t, RR, r \rangle$$

Where:

- n and r keep the same meaning
- t denotes the duration of the rule execution is a closed interval over $\mathbb{R}_{\geq 0}$
- RR is the set of resources used by the rule where each element is of the form $(rr,\ ra)$ where $rr \in ER$ is the resource name and ra is the resource amount consumed, specified as a closed interval on $\mathbb{R}_{\geq 0}$

4.2.5 Light Switch Example Version 2

The light switch example from Section 4.2.1 is extended with time annotations and resource annotations. The sample resources are memory and power. Memory has a maximum size of 16, 000 units and power has a maximum size of 100 units. The extended environment, as well as the extended machine specification are shown in Listing 4.3. In Listing 4.3, the rules specify that the execution of the first rule of the machine, rule $R1$, lasts between 4 and 10 time units. Furthermore, execution of rule $R1$ consumes 200 units of memory and 25 units of power. The semantics of this example, including sample runs, are given in Section 4.3.4. Listing 4.3 introduces the special *else rule*, which is enabled when no other rule is enabled. The else rule is used to prevent the machine from terminating when no other rule is enabled.

4.2.6 Hierarchical Composition

In complex systems, structuring mechanisms are required to partition large specifications into manageable blocks [6]. The partitioning enables bottom-up or top-down construction of specifications and creates opportunities for reuse. Furthermore, modularity enables separation of concerns and can help mitigate verification complexity [4]. The composition mechanisms included in the TASM language are based on the XASM language [10]. In the XASM language, an ASM can use other ASMs in rules in two different ways - as a *sub* ASM or as a $function$ ASM. A *sub* ASM is a machine that is used to structure specifications hierarchically, similar to a *Turbo ASM* [42]. A $function$ ASM is a machine that takes a set of inputs and returns a single value as output, similarly to a function in programming languages, and similar to an ASM *macro* [42]. These two concepts enable abstraction of specifications by hiding details inside of auxiliary machines. In the TASM language, the definition of a sub machine is similar to the previous definition of machine ASM given in Section 4.2.1:

$$SASM = \langle n, MV, CV, R \rangle$$

Where n is the machine name, unique in the specification, and the other tuple

Listing 4.3 Light switch example version 2 – time and resource annotations

```
  ENVIRONMENT:

USER-DEFINED TYPES:
  light_status  := {ON, OFF};
  switch_status := {UP, DOWN};

RESOURCES:
  memory    := [0, 16000];
  power     := [0, 100];

VARIABLES:
  light_status  light  := OFF;
  switch_status switch := DOWN;

--------------------------------

MAIN MACHINE:

MONITORED VARIABLES:
  switch;

CONTROLLED VARIABLES:
  light;

RULES:

  R1: Turn On
  {
    t         := [4, 10];
    memory    := 200;
    power     := 25;
    if light = OFF and switch = UP then
      light := ON;
}

  R2: Turn Off
  {
    t         := 6;
    memory    := 100;
    power     := 15;
    if light = ON and switch = DOWN then
      light := OFF;
  }

  R3: Else
  {
    else then
      skip;
  }
```

members keep the same definitions given in previous sections. The execution and termination semantics of a sub ASM are different than those of a main ASM. When a sub ASM is invoked, one of its enabled rules is selected, it yields an update set, and it terminates.

The definition of a function ASM is slightly different. Instead of specifying monitored and controlled variables, a function ASM specifies the number and types of the inputs and the type of the output:

$$FASM = \langle n, IV, OV, R \rangle$$

Where:

- n is the machine name, unique in the specification
- IV is a set of named inputs (ivn, it) where ivn is the input name, unique in IV, and $it \in TU$ is its type.
- OV is a pair (ovn, ot) specifying the output where ovn is the name of the output and $ot \in TU$ is its type.
- R is the set of rules with the same definition as previously stated, but with the restriction that it only operates on variables in IV and OV.

A function ASM cannot modify the environment and must derive its output solely from its inputs. The only side-effect of a function ASM is time and resource consumption. A specification, $ASMSPEC$, is extended to include the auxiliary ASMs:

$$ASMSPEC = \langle E, AASM, ASM \rangle$$

Where:

- E is the environment
- $AASM$ is a set of auxiliary ASMs (both sub ASMs and function ASMs)

- *ASM* is the main machine

The auxiliary machines are purely syntactic construct to ease reuse and structuring of specifications. As Theorem 4.1 and Theorem 4.2 state, the hierarchical composition can be eliminated without modifying the semantics. A description of the semantics of the concepts introduced in this section is available in Section 4.3.

4.2.7 Light Switch Example Version 3

The light switch example is extended to illustrate the use of auxiliary machines. The example has been extended with a function machine called TURN_ON and a sub machine called TURN_OFF. Sample runs for this example are given in Section 4.3.6.

Listing 4.4 Light switch example version 3 – hierarchical composition

```
FUNCTION MACHINE:               |       SUB MACHINE:
  TURN_ON                       |         TURN_OFF
                                |
INPUT VARIABLES:                |       MONITORED VARIABLES:
  switch_status ss;             |         switch;
                                |
OUTPUT VARIABLE:                |       CONTROLLED VARIABLES:
  light_status ls;              |         light;
                                |
RULES:                          |       RULES:
                                |
  R1: Turn On                   |       R1: Turn Off
  {                             |       {
    t      := [4, 10];          |         t := 6;
    memory := 128;              |
    if ss = UP then             |         if switch = DOWN then
      ls := ON;                 |           light := OFF;
  }                             |       }
                                |
  R2: Else                      |       R2: Else
  {                             |       {
    else then                   |         else then
      ls := OFF;                |           skip;
  }                             |       }
```

The two modified rules of the main machine from Listing 4.3 are shown in Listing 4.5. The remainder of the specification remains unchanged from Listing 4.2 and Listing 4.3.

Listing 4.5 Light switch example version 3 – modified rules to use auxiliary machines

```
R1: Turn On
{
  t         := 1;
  if light = OFF and switch = UP then
    light := TURN_ON(switch); //uses function machine
}

R2: Turn Off
{
  memory    := 1024;
  if light = ON and switch = DOWN then
    TURN_OFF(); //uses sub machine
}
```

4.2.8 Parallel Composition

To enable specification of multiple parallel activities in a system, the TASM language allows parallel composition of multiple abstract state machines. Parallel composition is enabled through the definition of multiple top-level machines, called *main* machines, analogous to multiple agents in [42]. Formally, the specification $ASMSPEC$ is extended to include a set of main machines $MASM$ as opposed to the single machine ASM for the basic ASM specification of Section 4.2.1:

$$ASMSPEC = \langle E, AASM, MASM \rangle$$

Where:

- E is the environment
- $AASM$ is a set of auxiliary ASMs (both sub ASMs and function ASMs)
- $MASM$ is a set of main machines ASM that execute in parallel

The definition of a main machine ASM is the same as the definition given in Section 4.2.3. Other definitions also remain unchanged.

4.2.9 Light Switch Example Version 4

In version 4 of the light switch problem, the example is extended to include an extra main machine that operates in parallel. The extra machine is used to control the fan. Listing 4.6 gives the environment definition, including the resources and the extra variables corresponding to the fan control. The main machine for the light control is shown in Listing 4.7. The main machine for the fan control is shown in Listing 4.8. The fan control machine contains time and resource annotations. The semantics of the parallel execution, as well as the consumption of resources are given in Section 4.3.8.

Listing 4.6 Environment definition for resources and parallel composition

```
ENVIRONMENT:

USER-DEFINED TYPES:
  component_status  := {ON, OFF};
  switch_status     := {UP, DOWN};

RESOURCES:
  memory    := [0, 16000];
  power     := [0, 100];

VARIABLES:
  component_status  light        := OFF;
  switch_status     light_switch := DOWN;
  component_status  fan          := OFF;
  switch_status     fan_switch   := DOWN;
```

4.3 The Timed Abstract State Machine (TASM) Language: Semantics

The semantics of the TASM language are expressed using the notions of *step*, *state*, and *update set* introduced in Section 4.2.1. The TASM language extends the update set concept with time and resource consumption. Updates to environment variables are organized in *steps*, where each step corresponds to a *rule execution*. In this section, the terms *step execution* and *rule execution* are used interchangeably. A rule is *enabled* if its guarding condition, C, evaluates to the Boolean value *True*. The

Listing 4.7 Light control main machine definition for resources and parallel composition

```
MAIN MACHINE:
  LIGHT_CONTROL

MONITORED VARIABLES:
  light_switch;

CONTROLLED VARIABLES:
  light;

RULES:

  R1: Turn On
  {
    t           := [4, 10];
    memory      := 300;
    power       := 25;
    if light = OFF and light_switch = UP then
      light := ON;
  }

  R2: Turn Off
  {
    t         := 6;
    memory    := 100;
    power     := 15;
    if light = ON and light_switch = DOWN then
      light := OFF;
  }

  R3: Else
  {
    t := next;
    else then
      skip;
  }
```

Listing 4.8 Fan main machine definition for resources and parallel composition

```
MAIN MACHINE:
  FAN_CONTROL

MONITORED VARIABLES:
  fan_switch;

CONTROLLED VARIABLES:
  fan;

RULES:

  R1: Turn On
  {
    t         := [1, 8];
    memory    := 100;
    power     := 35;
    if fan = OFF and fan_switch = UP then
      fan := ON;
  }

  R2: Turn Off
  {
    t         := 2;
    memory    := 200;
    power     := 25;
    if fan = ON and fan_switch = DOWN then
      fan := OFF;
  }

  R3: Else
  {
    t = next;
    else then
      skip;
  }
```

update set for the i^{th} step, denoted U_i, is defined as the collection of all updates to controlled variables for the step. An update set U_i will contain 0 or more pairs *(cv, v)* of assignments of values to controlled variables. A *run* of a basic ASM is defined by a sequence of update sets.

4.3.1 Update Set

In TASM, when a machine executes a step, the update set that is produced contains the duration of the step, as well as the amounts of resources that are consumed during the step execution. The special symbol $\perp$ is used to denote the absence of an annotation, for either a time annotation or a resource annotation. Update sets are extended to include the duration of the step, $t \in \mathbb{R}_{\geq 0} \cup \{\perp\}$ and a set of resource usage pairs $rc = (rr, rac) \in RC$ where rr is the resource name and $rac \in \mathbb{R}_{\geq 0} \cup \{\perp\}$ is a single value denoting the amount of resource usage for the step. If a resource is specified as an interval, rac is a value non-deterministically selected from the interval.

The symbol TRU_i is used to denote the timed update set, with resource consumptions, of the ith step of a machine, where t_i is the step duration, RC_i is the set of consumed resources, and U_i is the set of updates to variables:

$$TRU_i = (t_i, RC_i, U_i)$$

The structure of the update set is explained in the following subsections by extending the update set presented in Section 4.2.1.

4.3.2 Time

When a time annotation is included in a rule, the specified time denotes possible duration of the update set, specified as relative time between steps. The total time of a run of a single machine is simply the summation of the individual step times over the run. The update set concept is extended to include the duration t_i of the update set:

$$TU_i = (t_i, U_i)$$

The set of variable updates, U, is unchanged from Section 4.2.1. A run of a machine that terminates after n steps becomes:

$$TU_1,\ TU_2,\ \ldots,\ TU_n$$

The state concept is also extended to reflect the value of time for the given state. While the time values in the update sets are relative to the previous steps, the time values in the state are absolute. A given run starts execution at time $t = 0$. The *Timed State*, TS_i, where gt_i denotes the *global time* is defined as:

$$TS_i = (gt_i,\ S_i)$$

The state S_i is unchanged from Section 4.2.1. Given this definition of timed state, the sequence of states for a run that ends after n steps:

$$TS_1,\ TS_2,\ \ldots,\ TS_n$$

The $\circ$ operator is extended for the new definitions of state and update set:

$$TS_i = TS_{i-1} \circ TU_i = (gt_{i-1}, S_{i-1}) \circ (t_i, U_i) = (gt_{i-1} + t_i, S_{i-1} \circ U_i)$$

For a run that ends after n steps, the total time of the run would be gt_n and would be defined as the summation of the step times over the run:

$$gt_n = \sum_{i=1}^{n} t_i$$

4.3.3 Resources

Update sets are also extended to reflect resource consumption at each step. Each update set is extended to include a set of resource usage pairs $rc = (rr,\ rac) \in RC$

where rr is the resource name and rac is a single value denoting the exact amount of resource usage for the step. If a resource is specified as an interval, rac is a value taken from the interval. The symbol TRU_i is used to denote the timed update set, with resource usage, of the ith step, where t_i is the step duration, RC_i is the set of consumed resources, and U_i is the update set:

$$TRU_i = (t_i,\ RC_i,\ U_i)$$

The execution semantics are also extended to reflect resource usage. Because resources are limited quantities, if an executing ASM utilizes more than a resource's limits, execution halts. Execution is well-defined only if resource utilization falls below the boundaries of the available resources. Resource usage is slightly different than time in that the resource utilization for a given update set starts with the time of the previous update set and lasts through the rule completion. The consumption of a resource for an update set TRU_i lasts during an open interval $(gt_{i-1},\ gt_i]$. The state definition is also extended to reflect resource consumption:

$$TRS_i = (gt_i,\ SRC_i,\ S_i)$$

The sequence of states for a run that ends after n steps:

$$TRS_0,\ TRS_1,\ TRS_2,\ \ldots,\ TRS_n$$

For update sets with time and resources, the $\circ$ operator is defined as follows:

$$\begin{aligned} TRS_i = TRS_{i-1} \circ TRU_i &= (gt_{i-1}, RC_{i-1}, S_{i-1}) \circ (t_i, RC_i, U_i) \\ &= (gt_{i-1} + t_i, RC_i, S_{i-1} \circ U_i) \\ &= (gt_i, RC_i, S_i) \end{aligned}$$

For all gt in the open interval (gt_{i-1}, gt_i), the state TRS will be (gt, RC_i, S_{i-1}). This definition reflects the behavior that resource consumption will begin with the start of a rule execution and will last until the rule execution is finished.

Concurrent resource usage by multiple components is assumed to be additive. For example, if two components, x and y use the same resource A concurrently, where x uses amount a_x and y uses amount a_y, then the total resource usage amount is $a_x + a_y$. For the remainder of this chapter, the term *update set* refers to an update set of the TRU_i form and the term *state* refers to a state of the TRS_i form.

4.3.4 Light Switch Example Version 2 Revisited

The semantics of a basic specification with time and resource annotations can be illustrated using a sample run of the machine in Listing 4.3.

Three sample update sets for different initial conditions of variable values are shown below:

- Initial condition: ((light, OFF), (switch, UP))
  Update set: ((5, ((memory, 200), (power, 25)), ((light, ON))))
- Initial condition: ((light, ON), (switch, DOWN))
  Update set: ((6, ((memory, 100), (power, 15)), ((light, OFF))))
- Initial condition: ((light, OFF) (switch, DOWN))
  Update set: ((0, ((memory, 0), (power, 0)), $\emptyset$))

Formally, the behavior and state progression of the first set of initial conditions can be expressed as follows:

- $TRS_0 = (0, ((memory, 0), (power, 0)), ((light, OFF), (switch, UP)))$
- $TRU_1 = (5, ((memory, 200), (power, 25)), (light, ON))$
- $TRS_1 = TRS_0 \circ TRU_1 = (5, ((memory, 200), (power, 25)), ((light, ON), (switch, UP)))$

For all times gt in the open interval $[0, 5)$, the state TRS is $(gt, ((memory, 200), (power, 25)), S_0)$. The same logic can be applied to the other two sample runs.

Mapping the above update set and state on the time axis yields the following states over time:

- $t < 5$: (((memory, 200), (power, 25)), ((light, OFF), (switch, UP)))
- $t = 5$: (((memory, 200), (power, 25)), ((light, ON), (switch, UP)))
- $t > 5$: (((memory, 0)), ((light, ON), (switch, UP)))

The duration of 5 time units was non-deterministically selected from the interval [4, 10]. The sample run illustrates the execution semantics of interval duration. In the Listing 4.3, the rule duration is specified using an interval, `t := [4, 10];`. In a run, the update set contains a single figure for the duration of the step. For a run to be well-defined, the duration of the rule application for the step corresponding to this rule must be in the interval. For the sample run, the duration of 5 time units was selected non-deterministically and any value in the interval could have been used. The 200 units of memory resource are consumed from the beginning of the rule execution to the completion of the rule execution. This execution model is simple and intuitive and allows the specifier to explore various potential behaviors.

4.3.5 Hierarchical Composition

Semantically, hierarchical composition is achieved through the composition of update sets. A rule execution can utilize sub machines and function machines in its effect expression. Each effect expression produces an update set, and those update sets are composed together to yield a cumulative update set to be applied to the environment. To define the semantics of hierarchical composition, the semantic domain $\mathbb{R}_{\geq 0} \cup \{\perp\}$ is utilized. The special value $\perp$ is used to denote the absence of an annotation, for either a time annotation or a resource annotation.

Two composition operators are defined, $\otimes$ and $\oplus$, to achieve hierarchical composition. The $\otimes$ operator is used to perform the composition of update sets produced by effect expressions within the same rule:

$$\begin{aligned} TRU_1 \otimes TRU_2 &= (t_1, RC_1, U_1) \otimes (t_2, RC_2, U_2) \\ &= (t_1 \otimes t_2, RC_1 \otimes RC_2, U_1 \cup U_2) \end{aligned}$$

The $\otimes$ operator is commutative and associative. The semantics of effect expressions within the same rule are that they happen in parallel. This means that the time annotations will be composed to reflect the duration of the longest update set:

$$t_1 \otimes t_2 = \begin{cases} t_1 & \text{if } t_2 = \perp \\ t_2 & \text{if } t_1 = \perp \\ max(t_1, t_2) & \text{otherwise} \end{cases}$$

The composition of resources also follows the semantics of parallel execution of effect expressions within the same rule. The $\otimes$ operator is distributed over the set of resources:

$$\begin{aligned} RC_1 \otimes RC_2 &= (rc_{11}, \ldots, rc_{1n}) \otimes (rc_{21}, \ldots, rc_{2n}) \\ &= (rc_{11} \otimes rc_{21}, \ldots, rc_{1n} \otimes rc_{2n}) \\ &= ((rr_{11}, rac_{11}) \otimes (rr_{21}, rac_{21}), \ldots, \\ &\quad (rr_{1n}, rac_{1n}) \otimes (rr_{2n}, rac_{2n})) \\ &= ((rr_{11}, rac_{11} \otimes rac_{21}), \ldots \\ &\quad ((rr_{1n}, rac_{1n} \otimes rac_{2n})) \end{aligned}$$

In the TASM language, resources are assumed to be *additive*, that is, parallel consumption of amounts r_1 and r_2 of the same resource yields a total consumption $r_1 + r_2$:

$$rac_1 \otimes rac_2 = \begin{cases} rac_1 & \text{if } rac_2 = \bot \\ rac_2 & \text{if } rac_1 = \bot \\ rac_1 + rac_2 & \text{otherwise} \end{cases}$$

Intuitively, the cumulative duration of a rule effect will be the longest time of an individual effect, the resource consumption will be the summation of the consumptions from individual effects, and the cumulative updates to variables will be the union of the updates from individual effects.

The $\oplus$ operator is used to perform composition of update sets between a *parent* machine and a *child* machine. A parent machine is defined as a machine that uses an auxiliary machine in at least one of its rules' effect expression. A child machine is defined as an auxiliary machine that is being used by another machine. For composition that involves a hierarchy of multiple levels, a machine can play both the role of parent and the role of child. To define the operator, the subscript p is used to denote the update set generated by the parent machine, and the subscript c to denote the update set generated by the child machine:

$$\begin{aligned} TRU_p \oplus TRU_c &= (t_p, RC_p, U_p) \oplus (t_c, RC_c, U_c) \\ &= (t_p \oplus t_c, RC_p \oplus RC_c, U_p \cup U_c) \end{aligned}$$

The $\oplus$ operator is *not* commutative, but it is associative. The duration of the rule execution will be determined by the parent, if a time annotation exists in the parent. Otherwise, it will be determined by the child:

$$t_p \oplus t_c = \begin{cases} t_c & \text{if } t_p = \bot \\ t_p & \text{otherwise} \end{cases}$$

The distribution of the $\oplus$ operator over the set of consumed resources is the same as for the $\otimes$ operator:

$$\begin{aligned} RC_p \oplus RC_c &= (rc_{p1}, \ldots, rc_{pn}) \oplus (rc_{c1}, \ldots, rc_{cn}) \\ &= (rc_{p1} \oplus rc_{c1}, \ldots, rc_{pn} \oplus rc_{cn}) \\ &= ((rr_{p1}, rac_{p1}) \oplus (rr_{c1}, rac_{c1}), \ldots, \\ &\qquad (rr_{pn}, rac_{pn}) \oplus (rr_{cn}, rac_{cn})) \\ &= ((rr_{p1}, rac_{p1} \oplus rac_{c1}), \ldots, \\ &\qquad (rr_{pn}, rac_{pn} \oplus rac_{cn})) \end{aligned}$$

The resources consumed by the rule execution will be determined by the parent, if a resource annotation exists in the parent. Otherwise, it will be determined by the child:

$$rac_p \oplus rac_c = \begin{cases} rac_c & \text{if } rac_p = \bot \\ rac_p & \text{otherwise} \end{cases}$$

Intuitively, the composition between parent update sets and child update sets is such that the parent machine overrides the child machine. If the parent machine has annotations, those annotations override the annotations from child machines. If a parent machine doesn't have an annotation, then its behavior is defined by the annotations of the auxiliary machines it uses. These semantics enables the abstraction of timing analysis common in the real-time community [89] where program units, such as function calls, are annotated with timing bounds without analyzing the underlying behavior of the units. Furthermore, these semantics enable bottom up construction of specifications where the timing behavior can be defined by as the sum of the parts. The hierarchical composition semantics maintain the semantics of ASM where everything that occurs within a step happens in parallel. As in the case of ASM, conflicting updates to variables yield update set inconsistency.

Figure 4-2 shows a hierarchy of machines for a sample rule execution. Each numbered square represents a machine. Machine "1" represents the rule of the main machine being executed; all other squares represent either sub machines or function machines used to derive the update set produced by the main machine. Machine "3" is an example of a machine that plays the role of parent (of machine "7") and child (of machine "1").

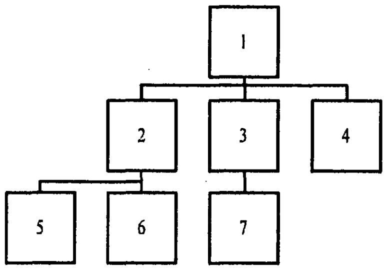


Figure 4-2: Hierarchical composition

Each machine generates an update set TRU_i, where i is the machine number. The derivation of the produced update set is done in a bottom-up fashion, where TRU_{ret} is the update set returned by the main machine:

$$\begin{aligned} TRU_{ret} = TRU_1 \oplus (\quad & (TRU_2 \oplus (TRU_5 \otimes TRU_6))\otimes \\ & (TRU_3 \oplus TRU_7)\otimes \\ & TRU_4) \end{aligned}$$

4.3.6 Light Switch Example Version 3 Revisited

The semantics of hierarchical composition are illustrated using the example from Listing 4.4 and Listing 4.5. Two sample runs are shown to illustrate the invocation of a sub machine and of a function machine. The first step of two sample runs are shown below:

- Initial environment: ((light, OFF), (switch, UP))

Update set: (1, ((memory, 128)), ((light, ON)))

- Initial environment: ((light, ON), (switch, DOWN))
  Update set: (6, ((memory, 1024)), ((light, OFF)))

The first sample run invokes the function ASM and obtains the step duration from the main ASM definition and the resource consumption from the function ASM. The second sample run obtains the variable updates and rule duration from the sub ASM and the resource consumption from the main ASM.

The first sample run can be detailed as follows:

- TRS_0 = (0, (($memory$, 0)), (($light$, OFF), ($switch$, UP)))
- Update set from function ASM: $FTRU_1$ = (5, (($memory$, 128)), $\emptyset$)
- Update set from main ASM: $RTRU_1$ = (1, (($memory$, $\perp$)), (($light$, ON)))
- Combined update set: $TRU_1 = RTRU_1 \oplus FTRU_1$ = (1 $\oplus$ 5, (($memory$, $\perp$)) $\oplus$ (($memory$, 128)), (($light$, ON)) $\cup$ $\emptyset$) = (1, (($memory$, 128)), (($light$, ON)))
- $TRS_1 = TRS_0 \circ TRU_1$ = (0, (($memory$, 0)), (($light$, ON), ($switch$, UP)) $\circ$ (1, (($memory$, 128)), (($light$, ON))) = (1, (($memory$, 128)), (($light$, ON), ($switch$, UP)))

The second sample run can be detailed as follows:

- TRS_0 = (0, (($memory$, 0)), (($light$, OFF), ($switch$, UP)))
- Update set from sub ASM: $STRU_1$ = (6, (($memory$, $\perp$)), (($light$, OFF)))
- Update set from main ASM: $RTRU_1$ = ($\perp$, (($memory$, 1024)), ($\emptyset$))
- Combined update set: $TRU_1 = RTRU_1 \oplus STRU_1$ = ($\perp \oplus 6$, (($memory$, 1024)) $\oplus$ (($memory$, $\perp$)), $\emptyset \cup$ (($light$, OFF))) = (6, (($memory$, 1024)), (($light$, OFF)))

- $TRS_1 = S_0 \circ TRU_1 = (0, ((memory, 0)), ((light, OFF), (switch, UP))) \circ (6, ((memory, 1024)), ((light, OFF))) = (6, ((memory, 1024)), ((light, OFF), (switch, UP)))$

The same operators can be used to detail the first sample run, which uses the function machine.

4.3.7 Hierarchical Composition and Expressivity

While hierarchical composition facilities are necessary in practice to enable reuse and to ease the management of complex models, hierarchical composition does not affect algorithmic expressivity. As Theorem 4.1 and Theorem 4.2 state, the hierarchical composition facilities of the TASM language could be eliminated without modifying the semantics of the language.

In the proof of Theorem 4.1, the following notation is used: the function machine is treated symbolically as "*f(params)*" where "*f*" is the name of the function machine and "*params*" is the list of parameters passed to the machine. The symbols "$\{exp \setminus val\}$", reused from programming language semantics, are used to denote "the resulting expression where symbols in *exp* are replaced by values in *val*". More specifically, it is used to replace the parameters in the function machine definition with the passed-in parameters in the function machine call. A parameterized version of a function machine, used in the proof of Theorem 4.1, is shown below:

$$\begin{aligned} FR_1 &\equiv if\ FG_1\ then\ out\_var \ := \ out\_exp_1; \\ FR_2 &\equiv if\ FG_2\ then\ out\_var \ := \ out\_exp_2; \\ &\vdots \\ FR_n &\equiv if\ FG_n\ then\ out\_var \ := \ out\_exp_n; \end{aligned}$$

Where *out_exp_i* represents an expression used to compute the output value of the function machine. With these definitions, Theorem 4.1 can be stated and proved.

Theorem 4.1. *For every machine that uses a function machine, there is an equivalent machine that does not use the function machine.*

Proof. The theorem is proved by construction, by providing an equivalent machine without the function machine. For a machine that uses a function machine, the function machine can occur in either the rule guard G_i or in the rule effect E_i. Both cases are considered separately:

- **Case 1: Function machine in rule guard**

If the function machine occurs in the rule guard, the guard will be of the form:

$$if\ g_{li}\ \diamond\ var = f(params)\ \diamond\ g_{ri}\ then\ E_i$$

Where $\diamond$ represents a logical connective, g_{li} represents the part of the guard to the "left" of the expression where the function machine occurs, and g_{ri} represents the part of the guard to the "right" of the expression where the function machine occurs. The function machine call could be part of a complex expression but the simplified version *f(params)* is used in the proof and can be easily generalized to any expression.

The equivalent machine can be constructed in the following way by replacing the rule where the function call occurs with n new rules that are constructed in the following way:

$$\begin{array}{l} if\ g_{li}\ \diamond\ var = \{out\_exp_1 \setminus params\}\ \diamond\ g_{ri}\ \wedge\ \{FG_1 \setminus params\}\ then\ E_i \\ if\ g_{li}\ \diamond\ var = \{out\_exp_2 \setminus params\}\ \diamond\ g_{ri}\ \wedge\ \{FG_2 \setminus params\}\ then\ E_i \\ \vdots \\ if\ g_{li}\ \diamond\ var = \{out\_exp_n \setminus params\}\ \diamond\ g_{ri}\ \wedge\ \{FG_n \setminus params\}\ then\ E_i \end{array}$$

It can be easily seen that the guards of each of these new rules will be enabled exactly when the function machine guards are enabled and when the guard of the original rule is enabled. And by replacing the invocation of the function machine with the return expression of the function machine definition ensures that the semantics aren't changed. Since evaluating rule guards does not consume time or resources, annotations occurring in the function machine can be discarded.

- **Case 2: Function machine in effect expression**

If the function machine occurs in the rule effect, the rule where the function machine call occurs will be of the form:

$$if\ G_i\ then\ e_{li};\ var := f(params);\ e_{ri};$$

Where e_{li} represents the part of the effect expression to the "left" of the function machine expression, e_{ri} represents the part of the effect expression to the "right" of the function machine expression, and *f(params)* represents the function machine expression. The function machine call could be part of a complex expression but the simplified version *f(params)* is used in the proof and can be easily generalized to any expression.

The equivalent machine can be constructed in the following way by replacing the rule where the function machine call occurs with n new rules in the machine definition:

$$\begin{gathered}
if\ G_i \wedge \{FG_1 \setminus params\}\ then\ e_{li};\ var := \{out\_exp_1 \setminus params\};\ e_{ri};\\
if\ G_i \wedge \{FG_2 \setminus params\}\ then\ e_{li};\ var := \{out\_exp_2 \setminus params\};\ e_{ri};\\
\vdots\\
if\ G_i \wedge \{FG_n \setminus params\}\ then\ e_{li};\ var := \{out\_exp_n \setminus params\};\ e_{ri};
\end{gathered}$$

It can be easily seen that the guards of each of these new rules will be enabled exactly when the function machine guards are enabled and when the guard of the original rule is enabled. And by replacing the invocation of the function machine with the return expression of the function machine definition ensures that the semantics aren't changed. For function machines occurring in effect expressions, time and resources can be consumed. How the annotations from the function machine are included in the equivalent "flattened" machine follows the rules of hierarchical composition of update sets described in Section 4.2.6.

These two cases can be generalized to any expression containing function machines by being applied to all expressions containing function invocations.

□

A parameterized version of a sub machine, used in the proof of Theorem 4.2, is shown below:

$$SR_1 \equiv if\ SG_1\ then\ SE_1;$$
$$SR_2 \equiv if\ SG_2\ then\ SE_2;$$
$$\vdots$$
$$FR_n \equiv if\ SG_n\ then\ SE_n;$$

Equipped with this definition, Theorem 4.2 can be stated.

Theorem 4.2. *For every machine that uses a sub machine, there is a equivalent machine that does not use the sub machine.*

Proof. This theorem is also proved by construction. Since sub machine invocations can only occur in effect expressions, only one case needs to be considered. An effect containing a sub machine call will be of the form:

$$if\ G_i\ then\ e_{li};\ SUB\_MACHINE(\ );\ e_{ri};$$

The construction of the equivalent machine is even simpler than it is for the proof of Theorem 4.1 since sub machines do not take in parameters. The equivalent machine, without the sub machine call, can be constructed in the following way:

$$if\ G_i \wedge SG_1\ then\ e_{li};\ SE_1;\ e_{ri};$$

$$if\ G_i \wedge SG_2\ then\ e_{li};\ SE_2;\ e_{ri};$$

$$\vdots$$

$$if\ G_i \wedge SG_n\ then\ e_{li};\ SE_n;\ e_{ri};$$

It can be easily seen that the guards of each of these new rules will be enabled exactly when the sub machine guards are enabled and when the guard of the original guard is enabled. The time and resource annotations from the sub machine are included in the "flattened" equivalent machine following the rules of hierarchical composition of update sets described in Section 4.2.6.

□

4.3.8 Parallel Composition

Because concurrency is an integral part of most real-time systems, the specification formalism must be able to specify concurrent behavior. In the abstract state machine world, this is achieved through multiple machines running in parallel. In the ASM literature, concurrency is termed *multi-agent* ASMs [47]. There are two varieties of multi-agent ASMs - synchronous and asynchronous. In the synchronous case, two or more machines execute a single step in parallel and the resulting update sets are checked for consistency, merged, and applied instantaneously to global state. In other words, for m machines executing concurrently, each executing n steps, all groups of step update sets, U_{ij}, must be consistent. U_{ij} denotes the i^{th} step of the j^{th} machine:

$$((U_{11}, U_{12}, \ldots, U_{1m}), \ldots, (U_{n1}, U_{n2}, \ldots, U_{nm}))$$

If a group of update sets is consistent for a given step, the updates sets are collected into a single update set and applied atomically to global state. The process is repeated for each step. In the asynchronous case, there is no prespecified order in which a machine executes a step. In fact, any ASM can perform any number of steps at a given time. This lack of ordering enables the system designer to define the exact semantics of parallel execution.

The semantics of parallel composition regards the synchronization of the main machines with respect to the global progression of time. The global time of a run, tb, is defined as a monotonically increasing function over $\mathbb{R}_{\geq 0}$. Machines execute steps that last a finite amount of time, expressed through the duration t_i of the produced update set. The *time of generation*, tg_i, of an update set is the value of tb when the update set is generated. The *time of application*, ta_i, of an update set for a given machine is defined as $tg_i + t_i$, that is, the value of tb when the update set will be applied. A machine whose update set, generated at global time tg_p, lasts t_p will be *busy* until $tb = tg_p + t_p$. While it is busy, the machine cannot perform other steps. In the meantime, other machines who are not busy are free to perform steps. This informal definition gives rise to update sets no longer constrained by step number, but constrained by time. Parallel composition, combined with time annotations, enables the specification of both synchronous and asynchronous systems.

The operator $\odot$ is defined for parallel composition of update sets. For a set of update sets TRU_i generated during the same step by i different main machines:

$$\begin{aligned} TRU_1 \odot TRU_2 &= (t_1, RC_1, U_1) \odot (t_2, RC_2, U_2) \\ &= \begin{cases} (t_1, RC_1 \odot RC_2, U_1) & \text{if } t_1 < t_2 \\ (t_2, RC_1 \odot RC_2, U_2) & \text{if } t_1 > t_2 \\ (t_1, RC_1 \odot RC_2, U_1 \cup U_2) & \text{if } t_1 = t_2 \end{cases} \end{aligned}$$

The operator $\odot$ is both commutative and associative. The parallel composition of resources is assumed to be additive, as in the case of hierarchical composition using the $\otimes$ operator:

$$\begin{aligned}
RC_1 \odot RC_2 &= (rc_{11}, \ldots, rc_{1n}) \odot (rc_{21}, \ldots, rc_{2n}) \\
&= (rc_{11} \odot rc_{21}, \ldots, rc_{1n} \odot rc_{2n}) \\
&= ((rr_{11}, rac_{11}) \odot (rr_{21}, rac_{21}), \ldots, \\
&\quad (rr_{1n}, rac_{1n}) \odot (rr_{2n}, rac_{2n})) \\
&= ((rr_{11}, rac_{11} \odot rac_{21}), \ldots \\
&\quad ((rr_{1n}, rac_{1n} \odot rac_{2n}))
\end{aligned}$$

The parallel composition of resources is assumed to be additive, as in the case of hierarchical composition using the $\otimes$ operator:

$$rac_1 \odot rac_2 = \begin{cases} rac_1 & \text{if } rac_2 = \bot \\ rac_2 & \text{if } rac_1 = \bot \\ rac_1 + rac_2 & \text{otherwise} \end{cases}$$

At each global step of the simulation, a list of pending update sets are kept in an ordered list, sorted by time of application. At each global step of the simulation, the update set at the front of the list is composed in parallel with other update sets, using the $\odot$ operator and the resulting update set is applied to the environment. Once an update set is applied to the environment, the step is completed and the global time of the simulation progresses according to the duration of the applied update set.

The concurrency semantics of the TASM language reduce to the concurrency semantics of synchronous and asynchronous multi-agent ASMs. For a TASM specification where all machine steps have the same duration $dt \neq 0$, the specification is essentially a synchronous multi-agent ASM specification with linear time progression. For a TASM specification where all machine steps have the same duration $dt = 0$, the specification is essentially an asynchronous multi-agent ASM specification. In TASM, time plays the role of delaying moves of a machine until the delay of the rule execution has elapsed and acts as a synchronization mechanism.

Parallel composition also introduces contention between machines for resource consumption. In the TASM language, no machine is preempted from using a resource. However, if the resource is exhausted, an exception is thrown and results in update set inconsistency. The shared resource model is simple and useful to model many resource types. Concurrent resource usage is additive. For example, if, in a time interval, two different machines use the same resource (in amounts r_1 and r_2 respectively), the total amount used would be $r_1 + r_2$. A more extensive application of the $\odot$ operator is shown to demonstrate the parallel composition of two update sets produced by two main machines that yield update sets with different durations:

- Update set by machine 1: $TRU1 = (t1, RC1, U1)$
- Update set by machine 2: $TRU2 = (t2, RC2, U2)$
- State when update set is produced: $TRS_i = (gt_i, RC_i, S_i)$
- $if\ t1 = t2$
  - Combined update set: $TRU = TRU1 \odot TRU2 = (t1, RC1 + RC2, U1 \cup U2)$
  - $TRS_{i+1} = TRS_i \circ TRU = (gt_i + t1, RC_i + RC1 + RC2, S_i \odot (U1 \cup U2)$
- $if\ t1 > t2$
  - Combined update set: $TRU_i = TRU1 \odot TRU2 = (t1, RC1 + RC2, U1)$
  - Combined update set: $TRU_{i+1} = TRU1 \odot TRU2 = (t2 \text{ - } t1, RC1, U2)$
  - $TRS_{i+1} = TRS_i \circ TRU_i = (gt_i + t1, RC_i + RC1 + RC2, S_i \odot U1)$
  - $TRS_{i+2} = TRS_{i+1} \circ TRU_{i+1} = (gt_i + t2, RC_i + RC2, S_{i+1} \circ U2)$
- $if\ t1 < t2$
  - Combined update set: $TRU_i = TRU1 \odot TRU2 = (t2, RC1 + RC2, U2)$
  - Combined update set: $TRU_{i+1} = TRU1 \odot TRU2 = (t1 \text{ - } t2, RC2, U1)$

- $S_{i+1} = TRS_i \circ TRU_i = (gt_i + t2,\ RC_i + RC1 + RC2,\ S_i \circ U2)$
- $S_{i+2} = TRS_{i+1} \circ TRU_{i+1} = (gt_i + t1,\ RC_i + RC1,\ S_{i+1} \circ U1)$

4.3.9 Light Switch Example Version 4 Revisited

In Listing 4.6 and Listing 4.8, the light switch example is further extended to illustrate the semantics of parallel composition. An extra main machine is added to represent the control logic for a fan, operating in parallel with the light. The logic for the fan and the light both utilize memory and power. For the initial environment (($light\_switch$, UP), ($light$, OFF), (fan_switch, UP), (fan, OFF)), the trace of update sets is shown below. Each machine will execute a single step that modifies the environment. The update sets for the step of each machine are shown below:

- Step 1 of machine `LIGHT_CONTROL`: (4, ((memory, 300), (power, 25)), ((light, ON)))
- Step 1 of machine `FAN_CONTROL`: (1, ((memory, 100), (power, 35)), ((fan, ON)))

This example shows how steps from different machines can take a different amount of time. The value of 4 time units for machine `LIGHT_CONTROL` and the value of 1 time units for machine `FAN_CONTROL` were taken non-deterministically from the intervals. The beginning of these steps happen at the same time, but the different durations illustrate the semantics of parallel composition. The time values of interest can be broken into five different intervals:

- $t < 1$: Execution of step 1 of both machines
- $t = 1$: Completion of step 1 of machine `FAN_CONTROL`
- $1 < t < 4$: Continued execution of step 1 of machine `LIGHT_CONTROL`
- $t = 4$: Completion of step 1 of machine `LIGHT_CONTROL`
- $t > 4$: Waiting for a change in the environment

The combined update sets for each time interval are shown below. The execution of both machines overlaps only in the interval t $\leq$ 1. In the other intervals, the behavior is that of individual machines. Updates to the environment are produced only at the end of the step:

- t < 1: (((memory, 400), (power, 60)), $\emptyset$)
- t = 1: (((memory, 400), (power, 60)), ((fan, ON)))
- 1 < t < 4: (((memory, 300), (power, 25)), $\emptyset$)
- t = 4: (((memory, 300), (power, 25)), ((light, ON)))
- t > 4: (((memory, 0), (power, 0)), $\emptyset$)

Formally, the state evolution can be tracked through the following stages:

- $TRS_0 = (1, ((memory, 0), (power, 0)), ((light, OFF), (fan, OFF)))$
- $TRU_{FAN\_CONTROL,1} = (1, ((memory, 100), (power, 35)), ((fan, ON)))$
- $TRU_{LIGHT\_CONTROL,1} = (4, ((memory, 300), (power, 25)), ((light, ON)))$
- $TRU_1 = TRU_{FAN\_CONTROL,1} \odot TRU_{LIGHT\_CONTROL,1} = (1, ((memory, 400), (power, 60)), ((fan, ON)))$
- $TRU_2 = TRU_{FAN\_CONTROL,1} \odot TRU_{LIGHT\_CONTROL,1} = (3, ((memory, 300), (power, 25)), ((light, ON)))$
- $TRS_1 = TRS_0 \circ TRU_1 = (1, ((memory, 400), (power, 60)), ((light, OFF), (fan, ON)))$
- $TRS_2 = TRS_1 \circ TRU_2 = (4, ((memory, 300), (power, 25)), ((light, ON), (fan, ON)))$

The time history of variable values and resource consumption for the run is also shown in Figure 4-3.

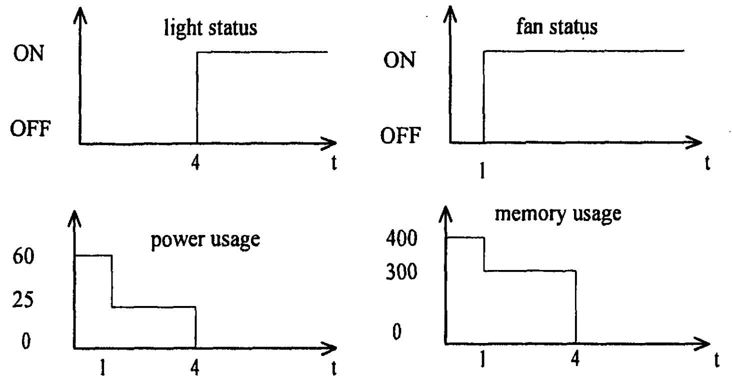


Figure 4-3: Time history of variable values and resource consumption

Operator	Signature	Meaning
$\circ$	*State* $\times$ *Update Set* $\rightarrow$ *State*	Used to apply an update set to the state
$\otimes$	*Update Set* $\times$ *Update Set* $\rightarrow$ *Update Set*	Used to combine two update sets generated through hierarchical composition, for update sets from different effect expressions
$\oplus$	*Update Set* $\times$ *Update Set* $\rightarrow$ *Update Set*	Used to combine two update sets generated through hierarchical composition, for update sets between a parent machine and a child machine
$\odot$	*Update Set* $\times$ *Update Set* $\rightarrow$ *Update Set*	Used to combine two update sets generated through parallel composition

Table 4.2: Update set combination operators

4.3.10 Summary and Other Extensions

From the point of view of the effects on the environment, there is no difference whether or not an update set is generated from a single main ASM or through multiple parallel main ASMs. The composition of main machines and the use of sub and function machines is indistinguishable to the environment. The environment only sees a single update set, that is produced at each "step" of the system. Once the composition has been achieved, the composed system behaves as a single main ASM with no composition. The difference occurs in the internal merging of update sets. In the case of hierarchical composition, that is, update sets produced by sub ASMs and function ASMs, the $\otimes$ and $\oplus$ operators are used to obtain the resulting update set from the use of auxiliary ASMs. For parallel composition, that is, multiple update sets produced by main ASMs, the $\odot$ operator is used to obtain the resulting update set that is to be applied to the environment. The update set operators are listed in table 4.2.

Termination Semantics

As described in Section 2.1, the types of systems targeted by this research are *reactive* real-time systems, that is, systems which continuously interact with their environment in an infinite loop fashion [164]. In this model, the environment could be modified outside of the machine's control. To enable this behavior the first extension to the ASM theory is to introduce the *Else Rule* construct. The *Else Rule* construct, denoted by the `else` keyword as a rule guard, is used to indicate that the machine will continue execution, even if no other rule is enabled. Furthermore, the use of the `skip` keyword is used to denote that an empty update set is produced but that execution should still continue.

A sample loop of such systems iterates through three stages. The first stage involves inferring the state of the environment, typically through sensors. The second stage involves taking some action, based on logic from the inferred state of the environment. The third and final stage involves affecting the state of the environment, typically through actuators. This loop will run continuously until the system is stopped by an outside source such as an operator or a failure. Applications using these types of loops are common in process controllers such as avionics and automobile electronics. The ASM metaphor, through the concepts of monitored variables, controlled variables, and steps reflects the behavior of reactive systems.

The assumption of termination when an empty update set is produced is not valid for reactive systems. The assumption may be valid for sequential algorithms, but, as can be observed in the light switch example, the controller should continuously monitor the state of the switch because the state of the switch could be altered outside of the machine's control.

Special Rule Durations

Since relative durations defines the underlying progression of time in the model, a special annotation can be used to specify that a given machine will "wait" until something meaningful happens in the environment. This annotation is used to denote that the machine will not execute any rules until something changes in the environment. This special annotation is the "`t := next`" construct. When a rule containing this time annotation is executed, the duration of the rule is "indeterminate" and the update set will be applied once another rule in another machine is executed.

Non-Determinism

While the TASM language does not include the *choose* construct from ASM, non-determinism is intrinsic to the TASM language. For example, time and resource annotations can vary non-deterministically. Input/Ouput non-determinism, in terms of assignments to variables, can occur in TASM if one or more rules are enabled simultaneously for a given step of a given machine. In this case, a rule is chosen non-deterministically from the enabled rules and it is executed. This type of non-determinism differs from ASM where multiple enabled rules are executed within the same step and the update sets are combined. In the TASM language, such semantics would be confusing since durations would have to be added. Furthermore, the ability to non-deterministically chose an enabled rule is convenient when modeling the environment to capture different simulation scenarios. The environment is inherently non-deterministic [208] and modeling this behavior is paramount to achieve realistic simulation scenarios.

The "Else" Rule

In the syntax of a TASM specification, the "Else" rule is used as shorthand notation for "a rule that is enabled is no other rule is enabled". While the simple keyword `else` is straightforward to write and understand, the special "Else" rule does not augment the semantics of the language. A machine definition containing an "Else" rule could be rewritten without the "Else" rule, without affecting the semantics. If a machine has n rules R_i, whose guards are G_i and where R_n is the "Else" rule, the guard of rule R_n is equivalent to the following guard:

$$G_n \equiv \neg(G_1 \ \vee \ \ldots \ G_{n-1})$$

Showing that these two guards are equivalent by definition because the "Else" rule is enabled whenever no other rules are enabled, which is exactly the definition of the negation of the disjunction of the guards of all the other rules. This substitution of the "Else" rule for a predicate over variables is important because in the rest of this thesis, the "Else" rule does not need to be treated differently than any other rule. Consequently, the "Else" rule is mentioned only where it is not evident from context that it could be replaced by a predicate over variables.

Internal State

In the presented discussion and examples, all variables are global. Extending individual machines with internal state enables encapsulation by limiting the scope of variables. A new section is added to each ASM definition, the "`INTERNAL VARIABLES`" section. This section is used to define the name and types of variables internal to the ASM.

Constructors

Reuse of specifications can be beneficial, especially for the specification of redundant systems. ASM definitions are extended with a "`CONSTRUCTOR`" section. The section lists variables whose values must be specified before a specific instance of the ASM specification can be used. This construct enables the creation of parameterized specifications to empower reuse.

The constructor concept introduces a new type of ASM, the template ASM. The template ASM is defined like any other ASM except that it contains an extra section, the `CONSTRUCTOR` section. Like a function ASM, the constructor section specifies the name and value types of arguments. Main machines can be defined based on template ASMs by using the machine name as constructor in the following fashion:

```
MAIN MACHINE:
  FAN_CONTROL := new FAN_TEMPLATE(OFF, 1)

...
```

Global Clock

Individual machines can obtain the current value of time by accessing the global clock. The global clock can be accessed through the special keyword "now" that returns a value denoting the current time, in the context of the querying machine. The value returned by "now" is the time value *before* the execution of a rule, based on the semantics of parallel composition from section 4.3.8.

Runs of Multi-Agent TASM Specifications

In [47], runs of synchronous and asynchronous multi-agent ASMs are described through ordering of agent moves. Synchronous multi-agent ASMs runs are defined through a total ordering of agent moves while runs of asynchronous multi-agent ASMs are defined through a partial ordering of agent moves. In the TASM language, time plays a key role in the synchronization of moves of agents. As mentioned in Section 4.3.8, depending on the nature of the time annotations, multi-agent TASM specifications can express both synchronous and asynchronous multi-agent ASMs. Consequently, the TASM language can be considered a more general model that can express both asynchronous and synchronous behavior, without modifying the underlying concurrency model.

The requirements of runs of multi-agent TASM specifications can be described in terms of partially ordered runs [115]. The ASM conditions on partially ordered runs contains 3 criteria – *finite history*, *sequentiality of agents*, and *coherence* [47] (p. 209). These three conditions also apply to runs of TASM specifications but the ordering relation is extended to include durations. The partially ordered set *(M, <)* of *moves* m is extended to include the timestamp of the move completion, t_c. The set is ordered with respect to t_c and with respect to a partial order for moves whose timestamps are the same. Using this ordering relation, the *sequentiality of agents* and the *coherence* conditions follow naturally.

The complete description of the TASM language is available in [185]. The concrete syntax of the TASM language as well as descriptions of the logical objects of the language, as implemented in the TASM toolset, is available in Appendix A.

4.4 Relation to Timed ASM

In [114], the authors present a specification and verification of the railroad crossing problem using a combination of ASM and the *currtime* external function (most recently called *now*). The algebra presented in [114] provides a general approach to timed system modeling. In order to demonstrate that TASM provides a more concise notation, the semantics of the TASM language are expressed using Timed ASM. In order to map a TASM specification into the Timed ASM language, two domains are introduced, namely $DTASM$ and $DASM$ to denote the domains of specifications expressed in the TASM language and the ASM language respectively. A function called *Desug* that maps a TASM specification into an ASM specification is also introduced. The "desugaring" function is defined for all individual elements of the TASM language (specifications, variables, types, rules, etc.) and maps the TASM elements into elements of the ASM language.

$$Desug: \ DTASM \rightarrow \ DASM$$

Each resource definition, $Rdef$, in the environment is desugared into a global shared dynamic function:

$$Desug[[\ Rdef\ ]] = \ \textbf{shared}\ Rdef$$

Type definitions, $Tdef$ get desugared into static finite domains:

$$\textit{Desug}[[\ Tdef\ ]] = \textbf{static domain}\ Tdef$$

Controlled and monitored variables inside of machines get desugared into nullary controlled and dynamic functions, respectively.

The desugaring of the rules is the most complex desugaring in the TASM language, because this is where time and resource utilization play a role. To illustrate the desugaring of rules, the following abstract syntax for a rule definition is utilized:

- $Rules = (R_i^+)$
- $R_i = (t_i\ r_i^*\ if\ cond_i\ then\ effect_i)$

In the TASM, the set of rules for a given machine is implicitly mutually exclusive. In the ASM language, the mutual exclusion is explicit. The desugaring introduces two variables, one to keep the time when the rule application will finish executing and one to denote that the machine is "busy" doing work. These two variables are denoted by $tcomp_{fresh}$ and $mbusy_{fresh}$. The $fresh$ underscore is used to indicate that the variable name is introduced by the desugaring and enforces that it does not clash with existing names. Both of these variables also desugar into controlled dynamic functions:

$$\begin{aligned} \textit{Desug}[[\ tcomp_{fresh}\ ]] &= \textbf{controlled}\ tcomp\ \text{initially}\ \ \textit{-1} \\ \textit{Desug}[[\ mbusy_{fresh}\ ]] &= \textbf{controlled}\ mbusy\ \text{initially}\ \ \textit{False} \end{aligned}$$

Conceptually, once a rule is triggered, a machine sets the $mbusy$ variable to $True$ and will not do anything until the rule duration has elapsed. Once the rule duration has elapsed, the machine will generate the appropriate update set atomically and will be free to execute another rule. The desugaring of a rule is expressed as:

```
Desug[[ Rule ]]    = Desug[[ ($t_i$ $r_i^*$ if $cond_i$ then $effect_i$) ]]
                   = **if/else if** $cond_i \wedge \neg mbusy_{fresh}$ **then**
                       $mbusy_{fresh} := True;$
                       $tcomp_{fresh} := now + getDuration(t_i);$
                       $r_{i_{fresh}} := getResourceConsumption(r_i);$
                     **else if** $now = tcomp_{fresh} \wedge mbusy_{fresh}$ **then**
                       $effect_i;$
                       $mbusy_{fresh} := False;$
                       $tcomp_{fresh} := -1;$
                       $r_{i_{fresh}} := 0;$
                       ...
```

The function *getDuration* is a macro that is created using the condition and the time annotation of the rule. It returns the duration of the rule by selecting a duration non-deterministically from the rule annotation. The introduction of the two auxiliary variables and the time conditions will guarantee that the machine will not produce any update sets and that no other rules will be enabled while the machine is executing a rule. This behavior is exactly the desired behavior to simulate "durative" actions. Function machines are desugared as macros and sub machines are desugared just like main machines and they are "inlined" inside the rule where they are invoked. The desugaring of the "*t := next*" construct is fairly straightforward albeit tedious. It involves caching the state at the beginning of the rule execution and creating an extra rule which compares the cached state to the current state. If there is a mismatch, the machine immediately resumes executing rules. If there is no mismatch, the machine simply waits until there is a mismatch.

The one area that remains to be formally specified is the execution semantics of resources. For each resource that is defined in the environment, an agent is created that is used to sum up all of the resources used by other agents. These new agents, symbolically depicted in Listing 4.9, are used to ensure that resource usage falls within the specified bounds.

Listing 4.9 Machine to compute resources

```
Agent $RESOURCE_i$
 controlled $last_{fresh}$ initially 0
 controlled $totalresource_{i_{fresh}}$ initially 0

  if $now = last_{fresh} + dt$ then
  $totalresource_{i_{fresh}} := sum(r_i)$
 else
  if $totalresource_{i_{fresh}} > resource_{i_{max}}$ then
  $RESOURCE\_EXHAUSTED$
```

The role of the sum macro is to sum up all of the resource annotations from executing agents. The *RESOURCE_EXHAUSTED* macro simply halts execution to note that a given resource has been exhausted.

4.5 Segue into Chapter 5

This chapter described the Timed Abstract State Machine (TASM) language through its syntax and semantics. The following chapter, Chapter 5, describes the types of automated analysis that can be performed on models expressed in the TASM language. More specifically, Chapter 5 describes how functional, timing, and resource consumption behavior of TASM models can be statically analyzed using the framework.

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 5

Static Analysis

This chapter describes the analysis of models that can be performed using the framework. The analysis that can be performed using the proposed framework include *Completeness and Consistency*, *Safety and Liveness*, *Execution Time*, and *Resource Consumption*. These four types of analyses are included in the four sections of this chapter. The analysis is performed with readily available analysis engines, namely the UPPAAL tool suite [157] and the SAT4J *SAT* solver [158]. The analysis is achieved by mapping TASM models to the input language of these engines. Summaries of the translation algorithms are given in this chapter but a complete version of the translation to *SAT* is available in Appendix B and a complete version of the translation to UPPAAL is given in Appendix C.

5.1 Functional Analysis: Completeness and Consistency

Consistency and completeness were identified as useful properties of specifications in [123] and in [125]. In the context of the specification of embedded systems, *completeness* of the specification is defined as the specification having a response for every possible class of inputs. In the same context, *consistency* is defined as the specification being free of contradictory behavior, including unintentional non-determinism [125].

Formal definitions of these properties, in the context of TASM specifications, are given in Section 5.1.2 and in Section 5.1.3. Traditionally, verifying these properties was accomplished manually by system specifiers, through inspection of specifications [66]. Because a specification is likely to evolve during the engineering lifecycle, the ability to verify these properties automatically can ease and shorten the analysis process [124]. Language-specific verification algorithms have been proposed in [123] in the context of the RSML requirement language and in [125] in the context of the SCR requirement language. In contrast, the analysis approach proposed in this chapter is not language specific and can be reused for other languages. The proposed approach achieves verification by translating specifications to formulas in propositional logic, formulating completeness and consistency as a Boolean satisfiability problem (SAT) [232], and automating the verification procedure by using a generally available solver, a SAT solver [175].

More specifically, the verification is achieved by mapping TASM specifications to Boolean formulas in Conjunctive Normal Form (CNF). The specified mapping is derived using the structural properties of the specification and does not require the generation of a global reachability graph, thereby avoiding the infamous state space explosion problem [125]. The proposed mapping could also be applied to specifications in other languages expressed using transition systems, such as ASM specifications, because the mapping does not consider the time or resource annotations of the TASM language. The mapping to Boolean formulas in CNF allows for automated verification using any SAT solver which conforms to the "DIMACS" format [158]. Using a standard input format provides flexibility in the choice of specific solver as optimizations and heuristics are constantly improving [229]. The mapping from TASM to a Boolean formula is achieved in such a way that consistency and completeness are expressed as unsatisfiability of the Boolean formulas. If the TASM specification is incomplete or inconsistent, the SAT solver will generate an assignment which makes the Boolean formula satisfiable. This assignment serves as the counterexample to exercise the incompleteness or inconsistency of the specification. Throughout this section, the "block form" of TASM from Equation 4.1 is used to define the concepts as they are introduced.

5.1.1 Related Work

The definition and automated verification of completeness and consistency of specifications were originally introduced in [123] and in [125]. In [123], the RSML language, a hierarchical state-based language, is used to express requirements. The language is automatically analyzed for completeness and consistency using an algorithm specifically developed for the RSML language. In [125], a similar approach is used for analysis of requirements expressed in the SCR language. These two approaches rely on special purpose algorithms for the efficient and automated analysis of consistency and completeness. Consequently, the proposed algorithms cannot be reused for other languages. In contrast, the approach proposed in this work utilizes a general purpose solver, a *SAT* solver. The proposed translation from TASM specifications to Boolean formulas in CNF can be reused for other specification languages. Furthermore, the use of a mature *SAT* solver guarantees that the analysis procedure is optimized since mature implementations of *SAT* solvers are generally available [229].

In the ASM community, various derivatives of the ASM language have been developed, including the ASM-SL language used in the ASM Workbench [59] and the Abstract State Machine Language (AsmL) used at Microsoft [116]. A mapping between ASM-SL and finite state machines, for the purpose of model checking, was proposed in [249]. A mapping between the AsmL language and finite state machines was proposed in [109, 110]. The mapping proposed in this thesis resembles the mappings proposed in these two approaches except that it ignores the effect of rule applications and does not need to generate a global reachability graph. The proposed mapping is concerned only with relationships between rule guards inside a single machine and hence produces a smaller state space than might be generated through a complete reachability graph.

5.1.2 Completeness

Informally, *completeness* is defined as the system specification having a response for every possible input combination. In the TASM world, for a given machine, this criteria means that a rule will be enabled for every possible combination of its monitored variables. The *monitored* variables are the variables in the environment which affect the machine execution. Formally, the disjunction of the rule guards of a given machine must form a tautology. The letter S is used to denote an instance of the SAT problem. The completeness problem can be expressed as a SAT problem in the following way:

For a given machine, for n rules:

$$S \equiv \neg \, (\, G_1 \, \vee \, G_2 \, \vee \, \ldots \, \vee \, G_n \,)$$

$$TASM = \begin{cases} \text{complete} & \text{if } S \text{ not satisfiable} \\ \text{incomplete} & \text{if } S \text{ satisfiable} \end{cases}$$

The completeness problem is casted as the negation of the disjunction so that counterexamples can be generated by the SAT solver. If S is satisfiable, all the assignments that make S satisfiable can be automatically generated by the SAT solver. If S is not satisfiable, the specification is complete.

Trivial cases happen when an individual rule guard represents a tautology. A specific example of a trivial case is the *else* rule. The *else* rule guarantees that the specification of a given machine is complete since the *else* rule will be enabled if no other rule is enabled.

Theorem 5.1. *Completeness is preserved through hierarchical composition using sub machines*

Proof. Per the definition of completeness, for a sub machine $\mathcal{SM}$ with m rules whose guards are of the form SG_j, if $\mathcal{SM}$ is complete, its rules form a tautology:

$$SG_1 \lor SG_2 \lor \ldots \lor SG_m \equiv T \tag{1}$$

Consider a machine $\mathcal{M}$ which uses sub machine $\mathcal{SM}$ in its effect expression, in rule R_p. Per the definition of completeness, if machine $\mathcal{M}$ is complete, its rules form a tautology:

$$G_1 \lor G_2 \lor \ldots \lor G_p \lor \ldots \lor G_n \equiv T \tag{2}$$

Per Theorem 4.2, an equivalent machine $\mathcal{M}'$ can be obtained by eliminating sub machine $\mathcal{SM}$ from rule R_p. The proof must demonstrate that $\mathcal{M}'$ is complete. Per the definition of completeness, it must be shown that the disjunction of the guards of the rules of machine $\mathcal{M}'$ form a tautology:

$$\begin{gathered} G_1 \lor G_2 \lor \ldots \lor \\ (G_p \land SG_1) \lor (G_p \land SG_2) \lor \ldots \lor (G_p \land SG_m) \lor \ldots \lor G_n \end{gathered} \tag{3}$$

Equation 3 can be rewritten by grouping the guard G_i and the guards SG_j and using the distributive law of $\land$ over $\lor$ [28]:

$$\begin{gathered} G_1 \lor G_2 \lor \ldots \lor \\ [G_p \land (SG_1 \lor SG_2 \lor \ldots \lor SG_m)] \lor \ldots \lor G_n \end{gathered} \tag{4}$$

By gathering terms and using the associativity of $\lor$, equation 4 can be rewritten:

$$[G_1 \lor G_2 \lor \ldots \lor G_n] \lor$$
$$[G_p \land (SG_1 \lor SG_2 \lor \ldots \lor SG_m)] \qquad (5)$$

Equation 5 can be expanded through the distribution law of $\lor$ over $\land$ [28] and gathering terms:

$$[G_1 \lor G_2 \lor \ldots \lor G_p \lor \ldots \lor G_n] \land$$
$$[(G_1 \lor G_2 \lor \ldots \lor G_n) \lor (SG_1 \lor SG_2 \lor \ldots \lor SG_m)] \qquad (6)$$

Given Equation 1 and Equation 6, it naturally follows that Equation 6 is a tautology since it can be reduced to the conjunction of two tautologies. Consequently, machine $\mathcal{M}'$ is complete and completeness is preserved through hierarchical composition using sub machines. The proof can be easily generalized to multiple sub machines within a given rule and across multiple rules.

□

Theorem 5.1 states that if a sub machine is complete and if a machine which uses the sub machine is also complete, then the equivalent machine without hierarchical composition (see Theorem 4.2) is also complete. This property is important because it implies that machines can be verified in isolation for completeness and those results still hold when combined hierarchically. These results are meaningful because it greatly reduces the complexity of the verification procedure since the derivation of the equivalent machine quickly leads to state explosion through exponential growth in the number of rules. Consequently, given Theorem 5.1, if all machines in a TASM specification are complete, then the specification is complete.

5.1.3 Consistency

Informally, for a state-based specification, *consistency* is defined as no state having more than one transition enabled at the same time [123]. The definition given in [125] is similar but is extended to include other properties of the specification such as syntactical correctness and type checking. The definition of consistency adopted in this approach is the same as in [123]. In terms of TASM specifications, this definition states that, for a given machine to be consistent, no two rules can be enabled at the same time. This definition will lead to a set of SAT problems to define consistency:

For a given machine, for each pair of rules R_i, R_j where $1 \leq i < j \leq n$:

$$S \equiv G_i \wedge G_j$$

$$TASM = \begin{cases} \text{consistent} & \text{if } S \text{ not satisfiable} \\ \text{inconsistent} & \text{if } S \text{ satisfiable} \end{cases}$$

This definition yields a set of $\begin{pmatrix} n \\ 2 \end{pmatrix}$ SAT problems. The individual SAT problems can also be composed into a single SAT problem. As for completeness, the SAT problem is defined in such a way that if the specification is not consistent, a counterexample is automatically generated. If S is satisfiable, all the assignments that make S satisfiable can be automatically generated by the SAT solver.

$$\begin{aligned} S \equiv\ & (G_1 \wedge G_2) \vee (G_1 \wedge G_3) \vee \ldots (G_1 \wedge G_n) \vee \\ & (G_2 \wedge G_3) \vee (G_2 \wedge G_4) \vee \ldots (G_2 \wedge G_n) \vee \\ & \qquad\qquad \vdots \\ & \qquad\qquad (G_{n-1} \wedge G_n) \end{aligned}$$

A trivial case occurs if there is only one rule. Other trivial cases happen if there are only two rules, one of which is a guarded rule and the other rule which is the special *else* rule. Before the *SAT* instances are generated, the *else* rule, if it exists, is removed from the machine specification being analyzed. It is important to note that consistency is a desirable property of specifications but not a requirement. For example, if the behavior of the environment is modeled, non-determinism can be introduced in the specification. But the remainder of the specification can be verified to be consistent even though the complete specification might not be consistent by choice. Similarly to Theorem 5.1, consistency of a machine can be verified in isolation and generalized to the complete specification.

Theorem 5.2. *Consistency is preserved through hierarchical composition using sub machines*

Proof. In this proof, the notation from the proof of Theorem 5.1 is reused. Given a consistent sub machine $\mathcal{SM}$ with m rules and a consistent machine $\mathcal{M}$ with n rules, it is shown that two tautologies follow from the definition of consistency.

Since machine $\mathcal{M}$ is consistent, for each pair of rules R_i, R_j of machine $\mathcal{M}$ where $1 \leq i < j \leq n$:

$$\neg(G_i \wedge G_j) \equiv \neg G_i \vee \neg G_j \equiv T \tag{1}$$

Since machine $\mathcal{SM}$ is consistent,, for each pair of rules SR_k, SR_l of machine $\mathcal{SM}$ where $1 \leq k < l \leq m$:

$$\neg(SG_k \wedge SG_l) \equiv \neg SG_k \vee \neg SG_l \equiv T \tag{2}$$

Per Theorem 4.2, an equivalent machine $\mathcal{M}'$ can be obtained by eliminating sub machine $\mathcal{SM}$. The proof must demonstrate that $\mathcal{M}'$ is consistent. Per the definition of consistency, it must be shown that the conjunction guards of the rules of machine

$\mathcal{M}'$ are invalid (that their negation forms a tautology). Since it is already known that the guards not affected by the sub machine call are already consistent with respect to each other, it is sufficient to show that the affected rules are consistent. If the sub machine call occurs in rule R_p, the guards of the affected rules in $\mathcal{M}'$ will be of the following form, per theorem 4.2, for $1 \leq k \leq m$:

$$(G_p \ \wedge \ SG_k) \tag{3}$$

When determining the consistency of the rules of machine $\mathcal{M}'$, two cases need to be considered. The first case involves the consistency of an affected rule with respect to an unaffected rule. Symbolically, it involves showing that Equation 4 forms a tautology, for $1 \leq k \leq m$, $1 \leq i \leq n$, and $i \neq p$:

$$\neg((G_p \ \wedge \ SG_k) \ \wedge \ G_i) \tag{4}$$

Equation 4 can be expanded using DeMorgan's Laws [28] and the terms can be rearranged using the associativity of $\vee$:

$$((\neg G_p \ \vee \ \neg G_i) \ \vee \ \neg SG_k)) \tag{5}$$

Given Equation 1, it is obvious that Equation 5 is a tautology. The second case involves the consistency of two modified rules with respect to one another. Symbolically, it involves showing that Equation 6 forms a tautology, for $1 \leq k < l \leq m$:

$$\neg((G_p \ \wedge \ SG_k) \ \wedge \ (G_p \ \wedge \ SG_l)) \tag{6}$$

Equation 6 can be expanded using DeMorgan's Laws [28] and the terms can be rearranged using the associativity of $\vee$:

$$(\neg G_p \vee (\neg SG_k \vee \neg SG_l)) \tag{7}$$

Given Equation 2, it is obvious that Equation 7 is a tautology. Consequently, machine $\mathcal{M}'$ is consistent and consistency is preserved through hierarchical composition using sub machines. The proof can also be easily generalized to multiple sub machines within a given rule and across multiple rules.

$\square$

5.1.4 Mapping to *SAT*

To implement the automated verification of completeness and consistency of TASM models in the TASM toolset, rule constraints are translated to Boolean formulas and verified using the SAT4J *SAT* solver. In order to translate TASM specifications to *SAT* , each variable included in the rule guards must be reduced to finite domains and mapped to Boolean propositions. The complete translation approach is detailed in Appendix B and the algorithm is summarized below:

1. Create problem instance S depending on the property to be checked (consistency or completeness), as explained in Section 5.1.2 and in Section 5.1.3
2. Replace function machine calls with extra rules, as explained in Section B.2.1
3. Replace symbolic right-hand sides with values from the chosen configuration, as explained in Section B.2.4
4. Reduce integer variables to user-defined type variables, as explained in Section B.2.3
5. Iterate through all monitored variables and create *at least one* clauses and *at most one* clauses, as explained in section B.2.2

6. Convert problem formulation S to conjunctive normal form and create the full SAT instance, as explained in Section B.2.2

The details of each step of the translation algorithm are explained in Appendix B. Some restrictions are imposed on TASM specifications that can be mapped to a SAT instance. In the current implementation of the TASM toolset, specifications containing float variables cannot be mapped to SAT unless a simple interval reduction is possible, as explained in Appendix B. The input format of popular SAT solvers is standardized according to the "DIMACS" format and must be input in Conjunctive Normal Form (CNF) [158]. The resulting SAT problem is automatically analyzed using the open source SAT4J SAT solver [158]. The toolset also provides the capability to "export" the generated SAT problem, so that the problem can be analyzed and solved outside of the TASM toolset.

5.1.5 Example

In this section, an example of the translation algorithm and the verification of completeness is provided. The example is a machine definition of the production cell case study presented in Section 2.8.1. The specification is for the behavior of the "loader" component, which is the component of the system responsible for putting blocks on the feed belt. The machine specification, expressed in the TASM language, is shown in Listing 5.1.

For the verification of completeness, the translation to SAT , for initial conditions where "$number = 5$", yields 7 unique propositions:

Listing 5.1 Definition of the loader machine

```
R1: The feed belt is empty, put a block on it
{
  t     := 2;
  power := 200;

  if loaded_blocks < number - 1 and feed_belt = empty then
    feed_belt       := loaded;
    loaded_blocks   := loaded_blocks + 1;
    feed_begin      := True;
}

R2: This is the last block...
{
  t     := 2;
  power := 200;

  if loaded_blocks = number - 1 and feed_belt = empty then
    feed_belt       := loaded;
    loaded_blocks   := loaded_blocks + 1;
    feed_begin      := True;
    loader_done     := True;
}

R3: The feed belt is loaded, do nothing
{
  t     := next;

  if feed_belt = loaded and loaded_blocks < number then
    skip;
}
```

$$
\begin{aligned}
b_1 &: loaded\_blocks \ <= \ 3 \\
b_2 &: loaded\_blocks \ = \ 4 \\
b_3 &: loaded\_blocks \ >= \ 5 \\
b_4 &: feed\_belt \ = \ empty \\
b_5 &: feed\_belt \ = \ loaded \\
b_6 &: feed\_block \ = \ available \\
b_7 &: feed\_block \ = \ notavailable
\end{aligned}
$$

Once the mapping between TASM variable values and *SAT* Boolean propositions has been established, the rule guards, G_i, can be expressed in terms of the Boolean propositions. The completeness problem, S, is then constructed according to the definition of completeness:

$$
\begin{aligned}
G_1 &\equiv b_1 \ \land \ b_4 \ \land \ b_7 \\
G_2 &\equiv b_2 \ \land \ b_4 \ \land \ b_7 \\
G_3 &\equiv b_5 \ \land \ (b_1 \ \lor \ b_2) \\
S &\equiv \neg(G_1 \ \lor \ G_2 \ \lor \ G_3)
\end{aligned}
$$

The complete translation to *SAT* , in CNF, yields 13 total propositions:

$$\begin{aligned}
&S\ in\ CNF \begin{cases} (\neg b_7 \vee \neg b_4 \vee \neg b_1) & \wedge \\ (\neg b_7 \vee \neg b_4 \vee \neg b_2) & \wedge \\ (\neg b_1 \vee \neg b_5) & \wedge \\ (\neg b_2 \vee \neg b_5) & \wedge \end{cases} \\
&At\ least\ one\ clauses \begin{cases} (b_1 \vee b_2 \vee b_3) & \wedge \\ (b_4 \vee b_5) & \wedge \\ (b_6 \vee b_7) & \wedge \end{cases} \\
&At\ most\ one\ clauses \begin{cases} (\neg b_1 \vee \neg b_2 \vee \neg b_3) & \wedge \\ (b_1 \vee \neg b_2 \vee \neg b_3) & \wedge \\ (\neg b_1 \vee b_2 \vee \neg b_3) & \wedge \\ (\neg b_1 \vee \neg b_2 \vee b_3) & \wedge \\ (\neg b_4 \vee \neg b_5) & \wedge \\ (\neg b_6 \vee \neg b_7) & \end{cases}
\end{aligned}$$

The *SAT* problem resulting from the translation is relatively small and running it through the SAT4J solver yields a solution in negligible time. For this machine, the rule set is not complete. The TASM toolset uses the SAT4J solver to generate the set of counterexamples in which no rule is enabled. An assignment to propositions that makes the problem satisfiable is "$b_2 = true$, $b_4 = true$, $b_6 = true$" and all other propositions are assigned *false*. In terms of the TASM specification, the counterexample which is generated is the set "$loaded\_blocks = 4$, $feed\_belt = empty$, $feed\_block = available$". To check the consistency of the rule set for the "loader" machine, the same set of propositions is generated, but the set of clauses grows to 159. However, many of the clauses are redundant, due to the long form used for the conversion to CNF. Future work in tool development will improve the translation to CNF by removing redundant clauses. Nevertheless, the set of *SAT* problems can be verified to be unsatisfiable in negligible time. In other words, the rules of machine "loader" are consistent.

5.2 Functional Analysis: Model Checking

As mentioned in Section 2.1, the functional correctness of a system can be formulated as a set of liveness and safety properties [154]. Formal verification through model checking represents one of the big successes from the formal methods community because it provides an approach to verification which is fully automated and can generate a witness trace [71]. Safety and liveness can be verified using a model checking approach by formulating the properties as temporal logic formulas. As mentioned in Section 2.7, a model checking approach is composed of some automata variant as a specification formalism and a temporal logic for property specification [67]. In the proposed framework, the model checking of functional properties utilizes the UPPAAL tool suite [24] which is a toolset for the modeling and verification of timed automata.

In order to verify the safety and liveness properties of TASM specifications using UPPAAL , the TASM models need to be translated to the timed automata of UPPAAL [29]. UPPAAL is a suite of tools to analyze real-time systems and is composed of an editor, a verifier, and a simulator [24]. Because the UPPAAL tool suite contains a model checker, the UPPAAL verifier, the translation to UPPAAL can be leveraged to also verify timing properties of TASM specifications using a combination of temporal logic and observer automata [129]. Safety assertions and liveness properties can be formulated in the temporal logic of UPPAAL , a subset of Timed Computation Tree Logic (TCTL) [244], and analyzed using the UPPAAL verifier. The verification of timing properties is described in Section 5.3. The TCTL of UPPAAL contains facilities for specifying quantifiers over variables, which are applied to paths and to states along the paths. The path quantifiers include "`A`", which means "for all paths" and "`E`", which means "there exists a path". The state quantifiers include "`[]`", which means "for all states" and "`<>`", which means "there exists a state". The formula ϕ is a predicate over variable values. The various combinations of the path and state quantifiers are given below, including the special quantifier "`-->`". A detailed description of UPPAAL 's TCTL is provided in [24].

- `A []` ϕ: For all paths, ϕ holds in all states.
- `A <>` ϕ: For all paths, there exists a state where ϕ holds.
- `E []` ϕ: There exists a path where ϕ holds in all states.
- `E <>` ϕ: There exists a path where there exists a state where ϕ holds.
- ψ `-->` ϕ: In all paths, if ψ holds, ϕ will eventually hold at a later point in the path.

5.2.1 Mapping to UPPAAL

To implement the automated verification of safety, liveness, and timing properties of TASM specifications in the TASM toolset, TASM models are translated to UPPAAL 's timed automata. The translation of TASM to timed automata involves removing function machines, sub machines, and translating TASM variables to UPPAAL 's datatypes. The complete translation approach is detailed in Appendix C and the algorithm is summarized below:

1. For each main machine in the TASM model, remove hierarchical composition according to the rules of Theorem 4.1 and of Theorem 4.2
2. Translate the environment:
   (a) Discard resource definitions
   (b) Translate each user-defined type to a corresponding bounded integer type of UPPAAL , as explained in Table C.2.1
   (c) Translate each variable and corresponding datatype to the bounded integer type of UPPAAL , as explained in Table C.2.1
3. For each "flattened" main machine

(a) Create a timed automaton to represent the machine

(b) Create an initial urgent location called "pivot"

(c) For each rule R_i of the machine, add a branch from the "pivot" state according to the approach explained in Section C.2.2

(d) If the machine contains an "else" rule, add an extra branch according to the approach depicted in Section C.2.2

(e) For rule that contains the "`t := next`" annotation, build an urgent edge using an extra automaton and an urgent channel

The details of each step of the translation algorithm are explained in Appendix C. Some restrictions are imposed on TASM specifications that can be mapped to timed automata. In the current implementation of the TASM toolset, specifications containing float variables cannot be mapped to timed automata. The toolset also provides the capability to "export" the generated UPPAAL problem, so that the problem can be analyzed and solved outside of the toolset.

5.2.2 Example

The example from Section 5.1.5 is reused in this section to illustrate the translation from TASM to UPPAAL and the verification of safety and liveness properties using UPPAAL 's TCTL. The TASM specification given in Listing 5.1 is translated to UPPAAL 's timed automata, and the result is given in Figure 5-1. The automaton in Figure 5-1 has three locations, corresponding to the three rules of Listing 5.1. The "Loader_R2_go" channel is an urgent channel used to enforce the transition after a state change has occurred, corresponding to the "`t := next`" annotation. The datatypes of TASM are translated to the bounded integer datatype of UPPAAL . In the case of the user-defined types of TASM, the enumeration members are converted to integers, as is the case for the `feed_begin` variable.

Once a TASM specification has been translated to timed automata, the UPPAAL tool suite can be used to verify safety and liveness properties of the specified system.

For example, for the timed automaton of Figure 5-1, a safety property stating that "the loader will never stop loading blocks until it has loaded all the blocks" can be formulated in TCTL. Furthermore, the liveness property "eventually, the loader loads the total number of blocks" can also be formulated in TCTL. Both properties are shown below:

- `A [] loader_done == 1 imply loaded_blocks == number`
- `A <> loaded_blocks == number`

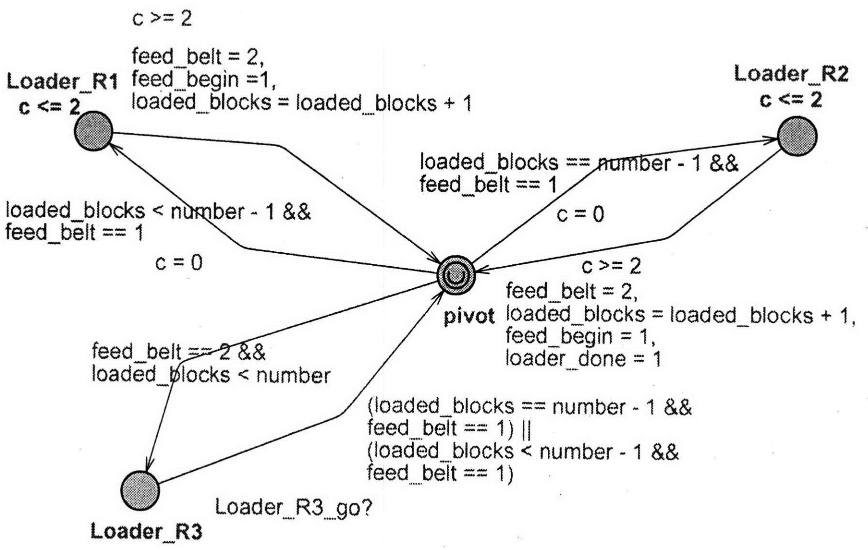


Figure 5-1: Timed automaton for Listing 5.1

5.3 Execution Time Analysis

This section provides a general approach to verify the minimum time and the maximum time that it takes for a TASM model to complete a path from an arbitrary state to another arbitrary state in the model. The time that can elapse in a TASM model is determined by the explicit time annotations contained in the specification. In the real-time system community, the terms Worst-Case Execution Time (WCET) and Best-Case Execution Time (BCET) [89] are used to denote properties of execution times of software and hardware implementations. In the real-time community, the BCET and WCET refer to execution times resulting from implementation artifacts where time passage is typically not explicitly stated and must be obtained through analysis. In this section and in the remainder of this thesis, the terms WCET and BCET are used, but in the context of a model where time passage is expressed explicitly. WCET and BCET are formulated as reachability problems in terms of the system states defined in the model. The execution time analysis approach can be used to perform a variety of time-related analysis, including schedulability analysis and system level analysis such as end-to-end latency. For example, in terms of system states, the traditional definition of the WCET of a task can be expressed as the maximum time that it takes for a task to reach the state *execution complete* from a state where the task is *ready for execution.* A similar formulation can be made for the BCET of a task.

Similarly to the verification of safety and liveness presented in Section 5.2, the verification of timing properties is achieved by mapping a TASM model to the input formalism of the UPPAAL analysis engine. For the verification of execution time, the timed automaton [5] formalism is used as the input language to define system states and system behavior. More specifically, the version of timed automata used is the extended version of the Alur-Dill formalism supported by the UPPAAL tool suite [29]. In the proposed approach, the timing analysis of system models is achieved automatically using the standard functionality of UPPAAL and modeling patterns – observer automata, the bounded liveness pattern, and temporal logic formulas [157].

The presented approach provides an approach to obtain WCET and BCET between *any* two system states, using an algorithm called *iterative bounded liveness.* The approach formulates execution time analysis as a reachability problem, which has been shown to be decidable for timed automata [7]. The use of observer automata removes the need to modify system models with extraneous annotations for the sole purpose of timing analysis, as is the case in [25], and in the bounded liveness pattern in [157]. In Section 5.3.3, the presented approach is illustrated through the analysis of a example problem depicting a simple scheduling problem. More complex examples are available in the case studies presented in Chapter 8.

5.3.1 Related Work

WCET analysis is an active research area in the real-time system community and sophisticated algorithms, models, and tools have been developed to analyze execution time of implementation languages for various hardware configurations. Thorough surveys of WCET techniques for programming languages and execution environments are available in [89] and in [150]. The output of these approaches can be used to annotate formal system models with timing properties, for the purpose of analyzing correctness with tools such as UPPAAL or Kronos [89]. The use of a formalism like timed automata can be used to analyze both timing correctness and functional correctness [157]. Furthermore, timed automata can be used for top-down analysis, to gain insight into system behavior before the system is implemented, when defects are typically cheaper to correct [37]. The approach to execution time analysis presented in this section can be considered a complement to the WCET analysis techniques of implementations, as performed in the real-time community. The *iterative bounded liveness* approach can be used in a top-down fashion before implementations exist, and can later be validated using bottom-up analysis provided by WCET approaches for software and hardware, once the system is implemented.

The use of high level system models for timing analysis has been performed in the context of statecharts [91] with compilation techniques. The approach presented in this work differs in that it relies on the explicit timing expressed in the model instead of translating the model to code to extract timing metrics. A similar approach was conducted in the context of Petri Nets [236] and in the context of priced timed automata [25]. The approach presented in this section differs in that it does not require modification of the system models for the purpose of timing analysis. Annotating the system model can inadvertently result in changes in the semantics of the model and clutters readability for the sole purpose of performing analysis. The use of observer automata is similar to the work on test case generation and time optimal test suites, summarized in [36], and in [128]. Using observer automata provides a flexible and non-intrusive way to analyze system models. The *iterative bounded liveness* approach is beneficial in that it provides flexible and reusable means of measuring minimum and maximum execution times between *any* two states of timed automata models, without modifying the system model. The approach is flexible and can be used to verify timing properties of system models such as schedulability and end-to-end latency.

5.3.2 Iterative Bounded Liveness

The approach to analyzing execution time is formulated using timed automata. In the rest of this section, it is assumed that a TASM specification has been translated to the timed automata of UPPAAL using the translation approach described in Appendix C.

The timed automaton formalism, also called Alur-Dill automata [5], extends finite state automata with a set of real-valued clocks to denote the passage of time. In a timed automaton, all transitions are instantaneous, but time elapses between transitions. Transition guards can contain predicates over clocks to enforce time passage before a transition is taken. State invariants are used to enforce upper bounds on the time passage in a given state. The timed automata used in UPPAAL extend Alur-Dill automata with Integer variables, Boolean variables, committed and urgent locations, and communication channels [24]. In UPPAAL 's timed automata, a location is equivalent to a state in Alur-Dill automata. *Urgent* locations are used to denote that time should not elapse in a location.

A brief review of the syntax and semantics of the timed automata of UPPAAL , combining terminology and notation from [29], [157], and [36] is provided. Formally, a timed automaton is a tuple $\langle L, l_0, C, V, I, E \rangle$, where:

- L is a set of locations
- $l_0 \in L$ is the initial location
- C is a set of real-valued clocks
- V is a set of variables

- I: $B(C) \rightarrow L$ is a mapping of simple clock constraints to locations, denoting location invariants
- $E \subseteq L \times G \times A \times L$ is a set of edges between locations where:
  - G is a predicate over variables V, simple clock constraints $B(C)$, and channel communications
  - A is a set of output actions, including clock resets, variable assignments, and channel communications

The semantics of a timed automaton are defined as a transition system over system states. A *state* $s \in S$ is a tuple $\langle\ l,\ \sigma,\ u\ \rangle$, where:

- $l \in L$ is an automaton location
- σ: $\mathbb{D} \rightarrow V$ is a mapping of values to variables
- u: $\mathbb{R}_{\geq 0} \rightarrow C$ is a valuation function for each clock in C

The initial state is $\langle\ l_0,\ \sigma_0,\ \{0\} \rightarrow C\ \rangle$, where all clocks are assigned the value 0. A *transition* is a relation $\mathcal{T} \subseteq S \times S$ whose members satisfy the following conditions:

- $\langle\ l,\ \sigma,\ u\ \rangle \longrightarrow \langle\ l',\ \sigma',\ (u \backslash r + t) \cup (\{0\} \rightarrow r)\ \rangle$ if $\forall\ t'$: $0 \leq t' \leq t \Rightarrow u + t \in I(l)$ and
- $\exists\ e \in E = \langle\ l,\ g,\ a,\ l'\ \rangle$ such that:
  - g is satisfied in $\langle\ l,\ \sigma,\ u\ \rangle$
  - the variable assignments in a yield σ' from σ
  - t is the amount of time elapsed in $\langle\ l,\ \sigma,\ u\ \rangle$
  - r is the set of clock resets in a

These definitions can be easily extended to networks of timed automata by using vectors. For a more detailed description of the syntax and semantics of UPPAAL 's timed automata, the reader is referred to [29].

Maximum and Minimum Execution Time

In the analysis of system specifications, the analysis of execution time is of special interest to understand the semantic properties of specifications that contain concurrent entities. The notation $a \rightarrow b$ denotes the "small-step" transition from a state $a \in S$ to a state $b \in S$. The relation $\mathcal{T}$ is used to denote the set of all "small-step" transitions $a \rightarrow b \in \mathcal{T} \subseteq S \times S$. The notation $a \mapsto b$ denotes the "big step" transition from state a to state b. The "big step" transitions can be defined in terms of "small-step" transitions:

$$a \mapsto b \equiv a \rightarrow s_1 \rightarrow \cdots \rightarrow s_n \rightarrow b$$
$$where: \ a \rightarrow s_1, s_n \rightarrow b \in \mathcal{T} \wedge$$
$$\forall\, 0 < i < n : s_i \rightarrow s_{i+1} \in \mathcal{T}$$

In other words, $\mapsto$ is the transitive closure of $\mathcal{T}$. The duration of a "small-step" transition, $t_{a \rightarrow b}$, is defined as the time $t \geq 0$ that can elapse during a transition $a \rightarrow b \in \mathcal{T}$. The duration of a "big-step" transition $t_{a \mapsto b}$ is defined in terms of durations of "small-step" transitions where $t_{a \mapsto b} \equiv (\Sigma_{i=1}^{n-1} t_{s_i \rightarrow s_{i+1}}) + t_{a \rightarrow s_1} + t_{s_n \rightarrow b}$. $t_{max_{a \mapsto b}}$ and $t_{min_{a \mapsto b}}$ are defined as:

$$t_{max_{a \mapsto b}} \equiv \{t'_{a \mapsto b} | \forall\, t_{a \mapsto b} : \ t_{a \mapsto b} \leq t'_{a \mapsto b}\}$$
$$t_{min_{a \mapsto b}} \equiv \{t'_{a \mapsto b} | \forall\, t_{a \mapsto b} : \ t_{a \mapsto b} \geq t'_{a \mapsto b}\}$$

When analyzing execution time of timed automata models, the properties of interest are BCET and WCET. *Execution time* is defined as the time, $t_{p_0 \mapsto p_1}$, that it takes to go from a state p_0 to a state p_1. The Best-Case Execution Time (BCET) is the lower bound of $t_{p_0 \mapsto p_1}$, that is, $t_{min_{p_0 \mapsto p_1}}$ and the Worst-Case Execution Time (WCET) is the upper bound of $t_{p_0 \mapsto p_1}$, that is, $t_{max_{p_0 \mapsto p_1}}$.

These properties can be analyzed using the UPPAAL tool suite and temporal logic formulas. The general problem of determining execution time of programs is undecidable because termination is undecidable, although approximations yield adequate results [89]. For specifications expressed using timed automata and verified using temporal logic, the reachability problem is decidable [7]. To verify execution time of timed automata, a combination of observer automata and the *bounded liveness* pattern [24] is used. *Bounded liveness* is a temporal logic formula pattern, combined with an augmentation of the model, which can be used to verify that execution time is appropriately bounded. The pattern is of the form $\phi_{max} \equiv$ "$A\ [\ ](b\ imply\ (z\ <=\ t))$" where b is a fresh Boolean variable and z is a fresh clock. Both b and z augment the timed automata model for the purpose of performing execution time analysis. The trick is for b to be true whenever the property p being checked holds. The clock z is reset when the property p begins to hold. Informally, the temporal logic formula states "the property p holds for at most t time units". The same idea is used, but it is grounds into an observer automaton instead of modifying the system model. The analysis also seeks to "find" the value of t, instead of "verifying" whether a model satisfies a t given a priori. To achieve this, the model is queried iteratively, to converge on the t corresponding to BCET or WCET.

Observer Automata

An observer automaton is a timed automaton which is not part of the system model, but which can be used to monitor certain properties of the system model. For example, observer automata have been used successfully in [36] to monitor coverage criteria with respect to test case generation. To verify execution time between two arbitrary states of the system model, the properties of the states that are monitored must be stated. The specification of properties is limited to predicates over variables and omit locations and clock values of the system model. In the context of the observer automaton, the time $t_{p_0 \mapsto p_1}$, means that there exists some trace $\langle\ *,\ \sigma,\ *\ \rangle \rightarrow \ldots \rightarrow \langle\ *,\ \sigma',\ *\ \rangle$ where $p_0 \subseteq \sigma$ and $p_1 \subseteq \sigma'$. This meaning does not affect the semantics of the model since it is part of the observer automaton and has no side-effect outside of the behavior of the observer automaton. The observer automaton is built using the concept of bounded liveness and using the state path to be verified. A sample observer automaton is shown in Figure 5-2, where z is a fresh clock variable, p_0 (a predicate over variable values) is the initial state from which to measure time, and p_1 (a predicate over variable values) is the final state to which time is measured. The observer automaton contains a Boolean variable b which is true (b==1) in all paths from p_0 to p_1. Furthermore, the clock z is added and reset for each transition out of p_0; the *OBSERVER_go* variable is an urgent channel, used to ensure that the only time that elapses in the observer automaton is time that elapses in the system model. Whenever a transition is enabled in the observer automaton, it is taken without delay. Location *q2* is marked urgent to ensure that no time elapses in that location.

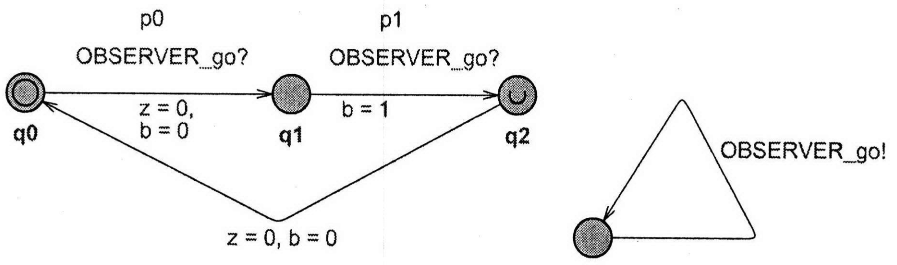


Figure 5-2: Observer automaton

Algorithm

In order to verify the execution time, the bounded liveness pattern of [157] is utilized iteratively. The temporal logic formula $\phi_{min} \equiv$ "$A\ [\ ](b == 1\ imply\ (z\ >=\ t))$" states that "the property p holds for at least t time units". Conversely, the formula $\phi_{max} \equiv$ "$A\ [\ ](b == 1\ imply\ (z\ <=\ t))$" states that "the property p holds for at most t time units". ϕ_{max} is used to obtain $t_{max_{p0 \mapsto p1}}$ and ϕ_{min} is used to obtain $t_{min_{p0 \mapsto p1}}$. $t_{max_{p0 \mapsto p1}}$ is obtained by iteratively verifying ϕ_{max} with increasingly large values for t until ϕ_{max} is satisfied. Conversely, $t_{min_{p0 \mapsto p1}}$ is obtained by iteratively verifying ϕ_{min} with smaller values for t until ϕ_{min} is satisfied. Because the UPPAAL model checker generates a counterexample when a temporal logic formula does not hold, the values of z given by the counterexample can be used as the value of t for the following iteration. The notation $\mathcal{TA}$ denotes the timed automata system model.

The notations $\phi_{min}(x)$ and $\phi_{max}(x)$ also denote ϕ_{min} and ϕ_{max} where t is substituted for x. The algorithm used to obtain the WCET is shown in Listing 5.2. The algorithm to obtain the BCET can be obtained by replacing ϕ_{max} by ϕ_{min} and WCET by BCET in Listing 5.2.

Listing 5.2 Iterative bounded liveness algorithm to calculate WCET

```
• Verify $\phi_{init}$ on timed automata model $\mathcal{TA}$
• If $\phi_{init}$ is satisfied, let $t_{est} = z$, using the value of clock $z$ stored in the UPPAAL
  simulator illustrating that $\phi_{init}$ is satisfied
• Loop until $\phi_{max}(t_{est})$ is satisfied
    – Verify $\phi_{max}(t_{est})$ on timed automata model $\mathcal{TA}$
    – if $\phi_{max}(t_{est})$ is not satisfied, let $t_{est} = z$, using the value of clock $z$ stored
      in the UPPAAL simulator
• $WCET = t_{est}$
```

The initial value of t used for the iteration can be obtained using the UPPAAL simulator and the simple reachability formula $\phi_{init} \equiv$ "$E <> \ oa.q2$" where $oa.q2$ is the state of the observer automaton that is reached when the path has been observed. If ϕ_{init} can be satisfied, there is at least one path from p_0 to p_1 from the initial state of the system model. Once $t_{min_{p_0 \mapsto p_0}}$ and $t_{max_{p_0 \mapsto p_1}}$ have been established, the UPPAAL simulator can be used to generate the sequence of steps that lead to $t_{min_{p_0 \mapsto p_1}}$ and $t_{max_{p_0 \mapsto p_1}}$ respectively. For $t_{min_{p_0 \mapsto p_1}}$, this can be achieved by setting $t = t_{min_{p_0 \mapsto p_1}} + 1$ in ϕ_{min} and reading the counterexample trace in the UPPAAL simulator. For $t_{max_{p_0 \mapsto p_1}}$, this can be achieved by setting $t = t_{max_{p_0 \mapsto p_1}} - 1$ in ϕ_{max} and reading the counterexample trace in the UPPAAL simulator. $t_{min_{p_0 \mapsto p_1}}$ is bounded from below by 0. If $t_{max_{p_0 \mapsto p_1}}$ is unbounded, ϕ_{max} will never be satisfied. Depending on the problem definition, a maximum value of t in ϕ_{max} should be agreed upon to determine that $t_{max_{p_0 \mapsto p_1}}$ is unbounded. It is important to note that a cycle in the state transitions will not lead to unbounded $t_{max_{p_0 \mapsto p_1}}$ because the clock z is reset on all transitions out of p_0. Unbounded $t_{max_{p_0 \mapsto p_1}}$ can occur purely as a side-effect of the properties of locations en route to p_1. For example, if a location on the path $p_0 \mapsto p_1$ has no location invariant, $t_{max_{p0 \mapsto p1}}$ could be unbounded. However, given the restrictions put on the models, this situation will never happen and $t_{max_{p0 \mapsto p1}}$ will converge, as explained in the following subsection.

Unbounded Delays and Convergence of Execution Time

In order for *iterative bounded liveness* to converge, the timed automata system model must not contain unbounded delays that can occur while the observer automaton of Figure 5-2 is in state *q1*. In timed automata, delays can be bounded either by setting a *location invariant* to limit how much time can elapse in a given location, or by having an *urgent edge* out of a given location. With these two conditions, delays are guaranteed to be bounded, given that urgent edges will become enabled within a bounded amount of time. While these restrictions might seem limiting, in the context of WCET analysis, all delays must be bounded, or a bound must be estimated as is the case of approximations for loop bounds in program analysis [89]. For systems that have unbounded delays, execution time analysis can occur for BCET, but WCET analysis would be pointless. The limitations on delays are similar to the restrictions in [235] where timed automata are restricted to be *output urgent* for the sake of testability.

5.3.3 Example: The Scheduling Problem

This subsection provides an illustrative example to illustrate the analysis of execution time. The example deals with the scheduling of the task graph shown in Figure 5-3. The task graph contains four tasks, each annotated with BCET and WCET figures. The arrows describe the precedence constraints for execution of the tasks. In order for the TASM system model to have unbounded delays, it is assumed that tasks will start executing as soon as the processor is free and when their dependencies (if any) have completed execution. This assumption is congruent with scheduling theory and prevents unbounded delays before the beginning of a task's execution.

The TASM model describing the scheduling problem is shown in Listing 5.3, in

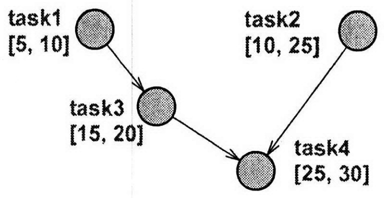


Figure 5-3: Task graph

Listing 5.4, and in Listing 5.5. Listing 5.3 shows the environment definition. A task can be in 3 possible states – *wait*, *execute*, and *done*. The *wait* state is used to denote a task that has not executed yet, the state *execute* is used to denote a task that is executing, and the state *done* is used to denote a task that has completed execution. The processor can be in 2 possible states – *free* and *busy*. The meaning of the processor states is self-evident. Listing 5.4 shows the definition of the scheduler that enforces the precedence constraints. It is interesting to note that the scheduler contains non-determinism, meaning that the set of rules are not consistent per the definitions given in Section 5.1.3. However, the non-determinism is introduced purposefully because given the problem definition, task1 and task2 can execute in any order because they do not have precedence constraints. Listing 5.5 shows the behavior of task 1. When the task is executing, it will take between 5 and 10 time units to complete, per the definition given in Figure 5-3. Each task is modeled as a main machine and TASM models for task 2, task 3, and task 4 are similar to Listing 5.5. The complete TASM model contains 5 main machines – 1 for the scheduler and 1 for each task.

The UPPAAL model, obtained through the translation algorithm described in Appendix C is shown in Figure 5-4, Figure 5-5, and Figure 5-6. Figure 5-4 shows the timed automaton for the TASK1 main machine of Listing 5.5. In the TASK1 automaton, location *pivot* is used to denote the initial location, location *TASK1_R1* is used to denote that the task is executing, corresponding to the execution of rule R_1 in the TASM main machine specification. The clock c is used to enforce the lower and upper bounds on the execution times of the task, through an invariant at location *TASK1_R1* and through a clock guard on the edge from the *TASK1_R1* location and the *pivot* location. The variable *proc* is used to signal that the processor is free

Listing 5.3 TASM environment for the scheduling problem

```
ENVIRONMENT:

USER-DEFINED TYPES:

task_status := {wait, exec, done};
proc_status := {free, busy};

VARIABLES:

task_status task1 := wait;
task_status task2 := wait;
task_status task3 := wait;
task_status task4 := wait;

proc_status proc  := free;
```

(*proc* == *1*) or busy (*proc* == *2*). There are 4 binary variables in the model, *task1*, *task2*, *task3*, *task4* to denote whether a given task has finished executing ($taskb ==$ *3*) or not ($taskn ==$ *2*). The TASK1 automaton has an urgent edge, enforced by the urgent channel *TASK1_else*, which ensures that a task starts executing as soon as it is capable, to avoid an unbounded delay before execution begins.

The goal of the example is to study the BCET and WCET of completing all the tasks with their precedence constraints. In the formulation of the problem, this property is equivalent to verifying the maximum and minimum amount of time for a path that goes from the state where none of the tasks have started executing ($task1 == 1$ && $task2 == 1$ && $task3 == 1$ && $task4 == 1$) to a state where all of the tasks have completed execution ($task1 == 3$ && $task2 == 3$ && $task3 == 3$ && $task4 == 3$). Following the convention of Section 5.3.2, the observer automaton encodes these two conditions as edge guards, and uses the clock z and the binary variable b to measure the time that elapses in location *q1* of the observer automaton of Figure 5-6.

Using the iterative bounded liveness approach, the BCET for all tasks to execute can be verified to be 55 time units and the WCET can be verified to be 85 units. It is trivial to verify that this result is correct because the BCET and WCET of the sequential execution of all the tasks is simply the summation of the individual BCETs

Listing 5.4 SCHEDULER main machine describing the behavior of the scheduler for the scheduling problem

```
R1: Execute task 1
{
  if task1 = wait and proc = free then
    task1 := exec;
    proc  := busy;
}

R2: Execute task 2
{
  if task2 = wait and proc = free then
    task2 := exec;
    proc  := busy;
}

R3: Execute task 3
{
  if task3 = wait and task1 = done and proc = free then
    task3 := exec;
    proc  := busy;
}

R4: Execute task 4
{
  if task4 = wait and task2 = done and task3 = done and proc = free then
    task4 := exec;
    proc  := busy;
}

R5:
{
  t := next;

  else then
    skip;
}
```

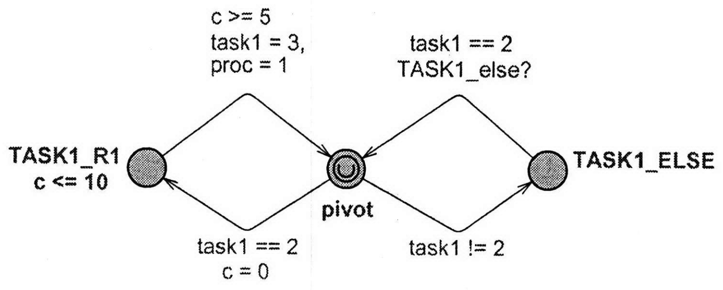


Figure 5-4: Timed automaton for the TASK1 main machine

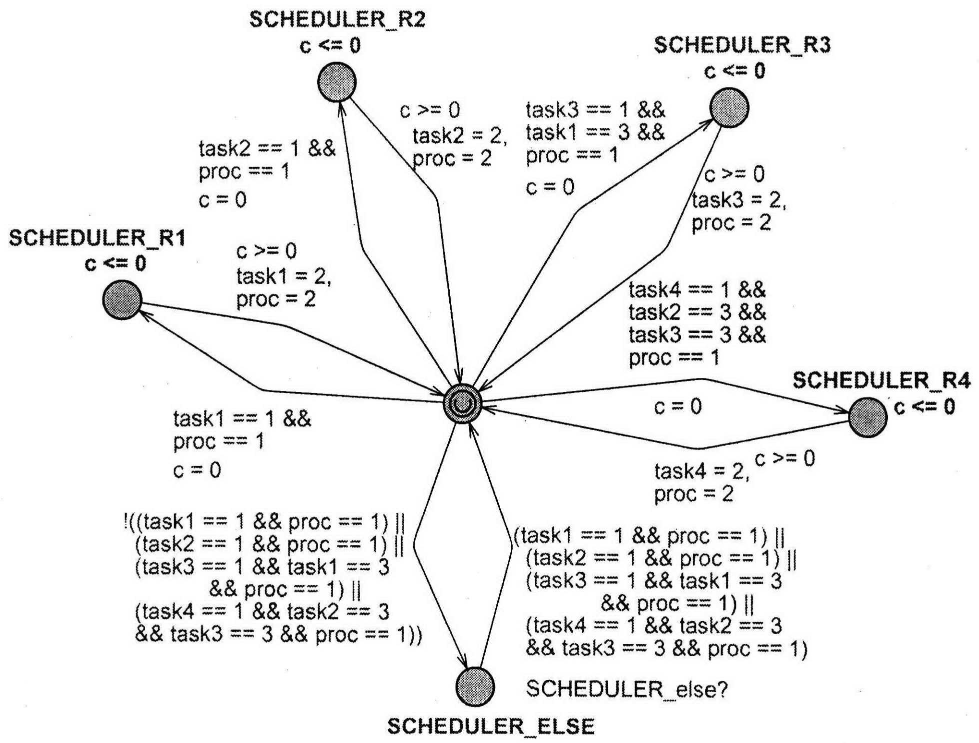
Listing 5.5 TASK1 main machine describing the behavior of task 1 of the scheduling problem

```
R1:
{
  t := [5, 20];

  if task1 = exec then
    task1 := done;
    proc  := free;
}

R2:
{
  t := next;

  else then
    skip;
}
```



SCHEDULER_R2
c <= 0
SCHEDULER_R3
c <= 0
SCHEDULER_R1
c <= 0
SCHEDULER_R4
c <= 0
SCHEDULER_ELSE
task2 == 1 && proc == 1
c = 0
c >= 0
task2 = 2, proc = 2
task3 == 1 && task1 == 3 && proc == 1
c = 0
c >= 0
task3 = 2, proc = 2
c >= 0
task1 = 2, proc = 2
task4 == 1 && task2 == 3 && task3 == 3 && proc == 1
c = 0
task1 == 1 && proc == 1
c = 0
task4 = 2, proc = 2
c >= 0
!((task1 == 1 && proc == 1) || (task2 == 1 && proc == 1) || (task3 == 1 && task1 == 3 && proc == 1) || (task4 == 1 && task2 == 3 && task3 == 3 && proc == 1))
(task1 == 1 && proc == 1) || (task2 == 1 && proc == 1) || (task3 == 1 && task1 == 3 && proc == 1) || (task4 == 1 && task2 == 3 && task3 == 3 && proc == 1)
SCHEDULER_else?

Figure 5-5: Timed automaton for the SCHEDULER main machine

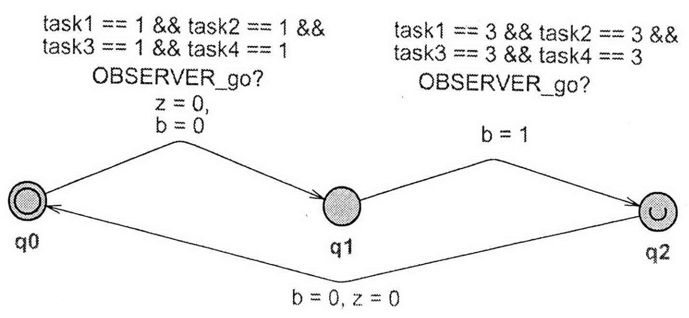


Figure 5-6: Observer automaton to analyze schedulability

and WCETs of each task. However, the purpose of this example was not to yield insight into the scheduling problem, but to give an illustrative example of the approach. The scheduler could also be extended to reflect a multi-processor architecture and the scheduling algorithm could be analyzed using the same observer automaton. More complex examples of observer automata and execution time analysis are available through the case studies presented in Chapter 8.

5.4 Resource Consumption Analysis

This section presents an approach to analyze the minimum and maximum amount of resources consumed by a TASM model, per the resource annotations. These minimum and maximum amounts are determined through an algorithm analogous to the algorithm to determine completeness and consistency presented in Section 5.1. The algorithm is implemented in the TASM toolset using a combination of the translation to SAT described in Appendix B and the translation to timed automata described in Appendix C.

5.4.1 Related Work

The analysis of resource usage, such as memory and stack usage has been performed in the context of programming languages [89, 119]. In the model based community, modeling of resources is gaining popularity [180], especially in the Quality of Service (QoS) community [252]. The approach presented in this section is unique in that it uses generally available solvers to calculate the best-case and worst-case resource consumption. Furthermore, the presented approach can calculate a safe upper bound and lower bound on resource consumption, without generating a global reachability graph, mitigating the state explosion problem. The proposed approach is flexible and can accommodate different levels of accuracy depending on the complexity of the problem at hand.

5.4.2 Approach

Because parallel resource usage is additive in the TASM language, the maximum amount of resources consumed will occur when the summation of the resource consumptions of parallel machines is at a maximum. To determine this maximum amount, the algorithm iterates through the rules of each machine and tries to find a set of rules for each machine that satisfies the following conditions:

For a TASM model with n main machines, for $1 \leq i \leq n$, where the subscript i denotes the i^{th} machine, and res_i denotes the amount of resources consumed by the i^{th} machine, and G_i denotes the guard of the i^{th} machine for the rule corresponding to the res_i resource consumption:

$$totres = \sum_{i=0}^{n} res_i \tag{5.1}$$

$$(\ G_1 \ \wedge \ G_2 \ \wedge \ \ldots \ \wedge \ G_n\ ) \text{ is satisfiable} \tag{5.2}$$

$$totres_{max} = \{totres \mid \forall\ totres' : \ totres' \leq totres\} \tag{5.3}$$

The state that satisfies Equation 5.2 is reachable (5.4)

The first condition simply restates the additive properties of the parallel consumption of resources. The second property states that the maximum amount of resources consumed must occur in a state where all rules consuming resources can be executed simultaneously. The third condition defines the maximum amount of resources consumed. The fourth condition states that the state that satisfies Equation 5.2 must be a reachable state.

Listing 5.6 Algorithm to determine maximum resource usage

```
• Remove hierarchical composition from all main machines $\mathcal{M}_i$ according to the
  approach explained in Theorem 4.1 and in Theorem 4.2
• $totres = -1$
• Loop over all sets of rules:
    – Select a set of rules that includes one rule from each machine $\mathcal{M}_i$
    – Calculate $totres'$ using Equation 5.1
    – if $totres' >$ totres then
        * if Equation 5.2 is satisfiable and Equation 5.4 holds then $totres = totres'$
• $totres_{max} = totres$
```

The first step of the algorithm calculates the "flattened" version of each main machine so that hierarchical composition is removed, to enable the direct comparison of rule guards. The algorithm loops over all sets of rules to try every combination of parallel rules to test the resource consumption of possible parallel behaviors. For each combination of rules, the resource consumption is calculated and the conjunction of the rule guards is checked for satisfiability. Satisfiability is a weaker notion than the logical properties introduced in Section 5.1 because there can be multiple states satisfying the disjunction of the rule guards. The first concern of the analysis is to determine whether Equation 5.2 is satisfiable. The following step of the algorithm concerns itself with determining whether the state that satisfies Equation 5.2 is a reachable state of the system. If reachability is not verified, the calculated version of $totres_{max}$ will yield a safe overapproximation. In other words, the calculated value provides a valid upper bound, but that upper bound might not be attainable in reality. For many systems, this approximation might be sufficient, in which case the reachability analysis can be skipped. However, if an exact measure is required, the reachability analysis will ensure that the algorithm yields an optimal value of $totres_{max}$. Clearly, the algorithm described in Listing 5.6 can be repeated for every resource in the specification. Furthermore, an analogous algorithm can be derived to calculate the minimum amount of consumed resources.

Implementation

The implementation of the algorithm in the TASM toolset follows the strategy of Section 5.1 and of Section 5.3. The implementation strategy uses mapping to both *SAT* and to UPPAAL 's timed automata to perform the analysis. The algorithm of Listing 5.6 does not specify how the sets of rules are assembled. The method used to select the set of rules can yield performance optimizations depending on the properties of the model. For example, the rules of individual machines can be sorted and $totres_{max}$ can be calculated using a breadth-first search approach. Other heuristics can be used to perform the calculation. In the TASM toolset, the approach sorts the rules of each machine and uses an exhaustive search. Possible optimizations could be performed and will be considered in future work. However, for the case studies of Chapter 8, the performance of the brute-force search proved adequate.

Verifying Equation 5.2 in the TASM toolset is achieved by translating the conjunction of the rule guards to *SAT* , following the approach presented in Appendix B. If there is a state s that satisfies Equation 5.2, the *SAT* solver will return the state s to the TASM toolset. The state s is returned to the toolset as a set of variables and associated values. The reachability analysis is implemented using the timed automata model and UPPAAL . The timed automata model can be obtained using the approach presented in Appendix C. If a timed automata model has already been generated for functional analysis or for execution time analysis, this model can be reused for the reachability analysis. The generation of the timed automata model needs to happen only once. To verify the reachability properties of state s, a simple reachability temporal logic formula can be used, with state s translated to the language of UPPAAL :

$$\texttt{E <>}\ s$$

If state s is reachable, the UPPAAL verifier will confirm that the formula holds.

5.4.3 Example

This section provides an example to illustrate the approach to verify minimum and maximum resource usage. The example reuses version 4 of the light switch example from Section 4.1.5, described in Listing 4.6, Listing 4.8, and Listing 4.7. Since this example does not contain hierarchical composition, it does not need to be flattened. Furthermore, since there are 3 rules in each main machine, there are 9 possible combinations of rule pairs that contain 1 rule from each machine. For each machine, the rules and corresponding memory consumptions are summarized below:

$$\begin{aligned}\texttt{LIGHT\_CONTROL}: &\langle R_1,\ \mathit{memory} = 300\rangle\ \langle R_2,\ \mathit{memory} = 100\rangle\ \langle R_3,\ \mathit{memory} = 0\rangle\\ \texttt{FAN\_CONTROL}: &\langle R_1,\ \mathit{memory} = 100\rangle\ \langle R_2,\ \mathit{memory} = 200\rangle\ \langle R_3,\ \mathit{memory} = 0\rangle\end{aligned}$$

Iterating through the possible pairs of rules, it is easy to see that the maximum memory consumption occurs when rule R_1 of machine LIGHT_CONTROL is executed and when rule R_2 of machine FAN_CONTROL is executed simultaneously. However, it must be determined whether the rule guards of these two rules can be enabled at the same time, that is, if the following formula is satisfiable:

```
(light = OFF and light_switch = UP) and
    (fan = ON and fan_switch = DOWN)
```

Name	Value	Rules	State
memory	500	R_1, R_2	*((light, OFF), (light_switch, UP), (fan, ON), (fan_switch, DOWN))*
bandwidth	60	R_1, R_1	*((light, OFF), (light_switch, UP), (fan, OFF), (fan_switch, UP))*

Table 5.1: Maximum resource usage

Name	Value	Rules	State
memory	0	R_3, R_3	*((light, ON), (light_switch, UP), (fan, ON), (fan_switch, UP))*
bandwidth	0	R_3, R_3	*((light, ON), (light_switch, UP), (fan, ON), (fan_switch, UP))*

Table 5.2: Minimum resource usage

Translating this formula to SAT and running it through the SAT solver shows that the formula is satisfiable with the state *((light, OFF), (light_switch, UP), (fan, ON), (fan_switch, DOWN))*. Encoding this state in a temporal logic formula and verifying the formula with the UPPAAL tool suite demonstrates that the state is reachable.

The same algorithm can be used to determine the maximum bandwidth usage for the model. Furthermore, the dual version of the algorithm can be used to determine the minimum memory and power usage for the model. The results of the analysis for the maximum resource usage are shown in Table 5.1 and the results of the analysis for the minimum resource usage are shown in Table 5.2.

5.5 Segue into Chapter 6

This chapter described the types of analysis that can be performed on TASM specifications using the proposed framework. More specifically, this chapter detailed how the completeness and consistency, safety and liveness, execution time, and resource consumption properties of TASM models can be analyzed automatically using the framework. The following chapter, Chapter 6, describes how two or more TASM models at different levels of abstraction can be meaningfully related. More specifically, Chapter 6 provides an approach to trace model features syntactically and integrates syntactic traceability with notions of semantic equivalence to establish a notion of semantic equivalence between the models. The presented approach enables bi-directional traceability of TASM models through levels of abstraction all the while ensuring semantic integrity under certain conditions.

Chapter 6

Bi-Directional Traceability

This chapter presents an approach to relate the syntax of two disparate TASM models. The proposed relationship, called *bi-directional traceability*, enables the tracking of model features throughout lifecycle phases and levels of abstraction, both for functional properties and for non-functional properties. Section 6.2.1 presents seven "standard" types of changes that can occur between two TASM models, as surveyed through modeling literature and experience with modeling. Each type of change is expressed as a syntactical mapping between the machines and rules of two TASM models. As presented in this chapter, traceability is a purely syntactical concept; however, in Section 6.2.2, for each proposed type of change, a set of correctness criteria is given to ensure that, if the criteria are met, the proposed change preserves the semantics of the original model. Section 6.3 provides an illustrative example to demonstrate the traceability approach combined with the use of the correctness criteria to ensure semantic equivalence throughout the change.

6.1 Related Work

The growing popularity of model-driven software engineering is yielding a new set of challenges for model management, model maintenance, and model evolution [168]. Since modeling typically happens at different levels of abstraction, often across lifecycle phases, the ability to relate disparate models to one another is becoming increasingly important to ensure consistency between models. For example, in the context of the Unified Modeling Language (UML), efforts have been exerted to define and enforce consistency between different diagrams [138]. Furthermore, the features of a model typically depend on a set of design decisions or assumptions. The ability to trace and visualize the effects of these features, and hence of associated decisions and assumptions, is important in complex system engineering [217]. Furthermore, traceability provides means to visualize and analyze the effects of changes to the specification, throughout the lifecycle of the system being engineered. However, traceability is, by definition, a syntactical concept, in the style of versioning systems, and provides no notion of semantic equivalence between the related artifacts. In the formal methods community, where models have precise semantics, theories of refinement have been developed to demonstrate semantic equivalence between two different models [174]. In these refinement approaches, the emphasis is put on correctness and imposes strict restrictions on system designers to ensure semantic correctness. In this chapter, a novel approach to model management is presented. The proposed strategy merges the benefits of syntactical traceability, for change management, and refinement correctness, for semantic integrity. The approach presented in this chapter provides an agile approach to relate models at different levels of abstraction and to relate models representing different aspects of the system such as functional behavior and timing behavior. The proposed traceability approach supplies benefits because it provides syntactic bi-directional traceability, augmented by a set of correctness criteria that can guarantee semantic integrity.

6.1.1 Syntactic Change Management

In the software engineering community, models of traceability have been developed for architecture models [94, 226] and for requirements [217]. In these approaches, change management and heterogeneous model integration is the key motivation. Furthermore, traceability models that cross lifecycle phases have been developed in [168] using the concept of connectors. Syntactic change management, in the context of programming languages, is widely used in the software engineering communities [78, 139].

In the context of programming language, the term "Software Configuration Management (SCM)" or "Version Control" are used to describe the set of tools and processes used to visualize program changes and to create software versions. A vast suite of tools are available to implement version control including open source offerings that include the Concurrent Versions System (CVS) [241]. In the software engineering community at large, notions of refinement correctness are typically overlooked and the focus is put on change management and syntactical mapping. The primary focus of traceability in the software engineering community is to visualize and control the changes that are made to product implementations, without concerning itself with the correctness of the changes.

In contrast, this chapter presents an approach to traceability that incorporates both syntactical traceability, to track changes, and a set of correctness criteria to maintain semantic integrity. Furthermore, in the engineering of real-time systems, different types of models are used, such as high level models, component models, and task graphs. The approach presented in this chapter enables traceability of model features throughout these disparate models, all the while ensuring semantic integrity under certain conditions.

6.1.2 Refinement Theory

The idea of software development being conducted in a controlled and provably correct fashion, in incremental steps, goes back to the days of Niklaus Wirth [250], and Edsger Dijkstra [80]. Since these seminal ideas, refinement theory has found widespread adoption and development in the formal methods community. In the formal methods community, the majority of refinement schemes revolve around two principles – the principle of "substituvity" and the principle of state equivalence. In the principle of "substituvity", the core idea revolves around the idea that a program could be replaced by another program and the change would be undetectable by the user [79]. In such a scheme, a refinement is deemed "correct" if the observable behavior of a program/model is undetectable after a refinement has occurred. The observations can take the form of input-output pairs, pre/post states and invariant preservation as can be found in the B method [2]. In the principle of state equivalence, the states of the two models to be related are enumerated, and a mapping is defined between the sets of states [43]. Furthermore, the semantic equivalence between two models can be established through a notion of trace equivalence through a subset relation and bisimulation, as can be found in the process algebra community [95, 131] and in the Input/Ouput Automata (I/OA) formalism [147]. The bisimulation proof method serves as the basis for many refinement approaches in the formal methods community and can express both "substituvity" or state equivalence, depending on what information the traces contain [227].

There has been a significant amount of theory developed around refinement, such as the refinement calculus [17], Morris' basis [174], and Roever's Data refinement [77]. These schemes aim to provide rigorous means by which refinement correctness can establish semantic equivalence. However, they do not address the syntactic nature of change management introduced by a refinement. The Abstract State Machine refinement approach is more general and can support many popular refinement schemes [43]. The ASM approach uses commuting diagrams as a mapping between states of interest and an equivalence notion ($\equiv$) between data in locations of interest in corresponding states. The approach proposed in this work is a subset of the general ASM refinement approach, by selecting a suitable set of criteria to establish basic correctness locally, without considering the complete semantics of the models. The correctness criteria proposed in this work do not preclude a more general notion of equivalence between models, as advocated in [43]. Formal approaches to refinement can be used in conjunction of the strategy proposed in this chapter, if desired.

From a practitioner's perspective, the motivations for establishing semantic equivalence between two models aim to reduce verification effort. In an ideal world, if verification was performed on a given model, the verification results would still hold in the refined model and the verification efforts would not need to be repeated. The philosophy of the approach presented in this chapter is to provide a set of correctness criteria which, if met, guarantee that verification results hold in the refined model.

6.2 Concepts

The concept of traceability establishes a mapping between two models. In the context of model-driven engineering, traceability typically happens between two models at different levels of abstraction, where the lower level model is assumed to be a *refinement* of the higher level model [168]. In general, the refined version of a model contains more details than the original version, although this property does not necessarily hold. For example, an optimization is a refinement which could remove details from an original model. This definition is also congruent with refinement concepts such as simulation relations in the formal methods community [174]. In this chapter, the concept of traceability and the concept of refinement correctness are differentiated. Traceability is defined as an invertible function between two models, mapping syntactical elements. The mappings can fall under different categories, depending on the differences between the two models. Refinement correctness can be established though a set of correctness criteria for the different types of syntactical mappings.

In the context of the TASM specification language, *traceability* is defined as a function between the rules of two models [202]. In the TASM language, rules are contained inside of machines. However, in the definition of traceability, the machine structure is ignored without loss of generality since rules can be renamed using the name of the machine as a prefix. The machine to which a rule belongs becomes important when specifying correctness criteria, but is irrelevant for the syntactical mappings.

Formally, traceability between model $\mathcal{M}_1$ and model $\mathcal{M}_2$ is defined as a partial function T over the set of rules X_1 of model $\mathcal{M}_1$ and the set of rules X_2 of model $\mathcal{M}_2$:

$$\mathrm{T} : \mathcal{P}(X_1) \rightarrow \mathcal{P}(X_2)$$

The partial function is defined over the power set of X_1 and the power set of X_2, as a function between arbitrary sets, to reflect the possibility that traceability does not have fixed arity and can be many-to-many. In the context of a partial function, the *domain of definition* is the set of elements in the domain for which the function is defined. In the context of a partial function, the *codomain of definition* is the set of elements in the image for which the function is defined. Two function operators are introduced, *ddef* and *codef*, which operate on T:

- *ddef*(T) $\triangleq$ domain of definition of T
- *codef*(T) $\triangleq$ codomain of definition of T

Although the function T is defined in a one-way fashion, bi-directional traceability can be achieved by taking the inverse of T. The ability to invert the function requires that the partial function be a bijection over *ddef*(T) and *codef*(T). Furthermore, to define the properties of T, a definition of a *partition* is given, where the empty set can be an element of the partition. Formally, a set P of subsets of X is a *partition* if it has the following properties:

- The union of the elements of $P = X$
- The pairwise intersection of the elements of $P = \varnothing$

With the definitions of *ddef*, *codef*, and *partition*, the formal definition of bi-directional traceability between two models can be formulated. The definition is given in terms of the properties of the function T:

- T is a bijection over *ddef*(T) $\rightarrow$ *codef*(T)
- *ddef*(T) must form a partition of X_1
- *codef*(T) must form a partition of X_2

The requirement that the function be a bijection over the domains of definition ensures that the function is invertible. This is a necessary condition for traceability to be bi-directional. The partition requirement on the domains of definition ensures that every rule in X_1 and X_2 is involved in one and only one mapping for each refinement level. This requirement is necessary when trying to ensure semantic equivalence between the two models using the correctness criteria.

6.2.1 Types of Refinements

In the previous section, traceability through levels of abstraction was established as a mapping between the rules of two TASM models. In this section, categories of mappings are defined so that the differences between the two models capture the rationale for the change. More specifically, this section defines seven categories of refinements, defined formally as elements of the function T, defined in the previous section, which share a common property and arity. Each category defines a conceptual type of refinement that can be traced bi-directionally.

The list of refinement types have been motivated by synthesis of refinement approaches as found in the literature [17, 174] and through experience with developing models using stepwise refinement. At a high level, in the context of TASM models, a *refinement* is defined as the addition, deletion, or modification of rules in a model $\mathcal{M}_1$, resulting in a *refined* model $\mathcal{M}_2$. The types of refinement described in this section attempt to express the motivations for performing a refinement. It is important to restate that traceability does not depend on a particular type of change since the types of changes are simply elements of the function T, that is, mappings between rules. But the categories are helpful syntactically, to convey the rationale behind the mapping, and semantically, when trying to establish semantic equivalence between the models. The categories are used in Section 6.2.2 where criteria are defined, for each category, to establish the correctness of the refinement. The refinement categories are expressed as ordered pairs of sets of rules. For example, the refinement described below denotes an arbitrary mapping between n rules of one model and m rules of another model:

$$\langle\{R_1,\ R_2,\ \ldots,\ R_n\},\{S_1,\ S_2,\ \ldots,\ S_m\}\rangle$$

The subscripts of the rules denote the ordinal of the rule in the ordered set. For certain categories, the ordering of rules is irrelevant but for other categories, the ordering is important. When the ordering matters, it is used to denote a sequence of execution starting with the first ordinal and terminating with the last ordinal.

Step Expansion

A step expansion refers to a type of refinement where a "step" in model $\mathcal{M}_1$ is refined into multiple steps in model $\mathcal{M}_2$. In the formal methods community, such a refinement is called a "non-atomic" refinement [79]. In the context of the TASM language, since a step is the execution of a rule, the expansion of a step implies a mapping between one rule and multiple other rules, a one-to-many relationship:

$$T_{sexp} \triangleq \langle \{R_1\},\ \{S_1,\ S_2,\ \ldots,\ S_m\} \rangle$$

In a step expansion refinement, the ordering of the rules S_i is important, especially when the correctness criteria are defined.

Step Contraction

The step contraction refinement is the dual of the step expansion refinement. A step contraction refinement refers to a type of refinement where multiple steps in model $\mathcal{M}_1$ are refined into a single step in model $\mathcal{M}_2$. The refinement is a many-to-one mapping:

$$T_{scon} \triangleq \langle \{R_1,\ R_2,\ \ldots,\ R_n\},\ \{S_1\} \rangle$$

As for the step expansion refinement, the order of the rules R_i is important.

Rule Expansion

A rule expansion refinement is a one-to-many mapping. In the context of a single rule, a rule expansion refinement is used to add or modify a time annotation, to add or modify resource annotations, to add more conditions to the rule guard, and to add extra effect expressions. The rule expansion refinement can also be used to expand a rule into multiple rules, resulting in a one-to-many mapping. Syntactically, the rule restriction refinement is similar to the step expansion refinement. The differences occur in the correctness criteria used to establish the semantic equivalence between the mapped rules. Conceptually, the step expansion refinement is meant to describe a refined sequential execution, where the extra rules are executed in sequence. The rule expansion refinement is meant to describe the expansion of the state, through the addition of variables to the model or by expanding the list of members in a user-defined type. The added state components can lead to added conditions in the rule guards and added assignments in the rule effect expressions.

$$T_{rres} \triangleq \langle \{R_1\},\ \{S_1,\ ,\ \ldots,\ S_m\} \rangle$$

Rule Contraction

Similarly to the step contraction refinement, the rule contraction refinement is also a many-to-one mapping. The rule contraction refinement is the dual of the rule expansion refinement. The rule contraction refinement is used to remove or modify a time annotation, to remove or modify resource annotations, to remove conditions from the rule guard, and to remove effect expressions. The refinement is also used to remove state components from the model, through the removal of variables or the removal of members of user-defined types. Removing variables can lead to a reduced number of rules as the number of items in the rule guards and in the rule effect expressions are reduced.

$$T_{rexp} \triangleq \langle \{R_1,\ \ldots,\ R_n\},\ \{S_1\} \rangle$$

Rule Addition

Rule addition refers to a type of refinement where behavior is added to a model, caused by expansion of the state space. In terms of the TASM language, this refinement corresponds to the addition of one or more rules to an existing set of rules, a zero-to-many mapping. The difference between the rule addition refinement and the rule expansion refinement concerns the correctness criteria. A rule addition refinement which adds p rules to a set of m rules would yield the following mapping:

$$T_{add} \triangleq \langle \{\ \}, \{S_{m+1}, \ \ldots, \ S_{m+p}\} \rangle$$

Rule Deletion

A rule deletion refinement is a refinement where a set of rules is removed from an existing set, caused by a reduction in the state space. The rule deletion refinement is the dual of the rule addition refinement, and it is defined as a many-to-zero mapping. A rule deletion refinement which removes p rules from a set of m rules would yield the following mapping:

$$T_{del} \triangleq \langle \{R_1, \ \ldots, \ R_{m-p}\}, \ \{\ \} \rangle$$

Any

While the three categories of refinements defined above, with their associated dual, represent common refinement types as surveyed through the literature and identified through modeling experience, it is possible that other types of refinements are necessary. Furthermore, since traceability is not dependent on the types of refinements, it is important not to restrict the usability of the traceability features by requiring the strict use of refinement types. Because of these motivations, a "wild card" type of refinement is defined, to define traceability without intent. This wild card, called an "any" refinement, is simply a generic many-to-many mapping between the rules of two models:

$$\langle \{R_1,\ R_2,\ \ldots,\ R_n\}, \{S_1,\ S_2,\ \ldots,\ S_m\} \rangle$$

Complete Traceability Relationship

These seven types of refinements are also complemented by the identity refinement, T_{id}, which maps a rule from model $\mathcal{M}_1$ to an identical rule in model $\mathcal{M}_2$. Conceptually, the identity refinement is simply a special case of all seven types of refinements; consequently, it is not described as a separate refinement, but it is introduced as special notation that is useful when establishing semantic equivalence. Given these relationships, define the complete traceability relationship between a $\mathcal{M}_1$ and a refined model $\mathcal{M}_1$ can be defined, through the partial function T:

$$\begin{aligned} \mathrm{T} &= \mathbb{T}_{sexp} \cup \mathbb{T}_{scon} \cup \mathbb{T}_{rres} \cup \\ &\quad \mathbb{T}_{rexp} \cup \mathbb{T}_{add} \cup \mathbb{T}_{del} \cup \mathbb{T}_{any} \cup \mathbb{T}_{id} \end{aligned}$$

Where each $\mathbb{T}_n$ is a set whose elements are refinements of type T_n, corresponding to the types of refinements defined in previous subsections. Given the definition of T, the definition of the various categories of refinements, and the relationship between the categories and the function T, a syntactical basis for bi-directional traceability of software models is established. In Section 6.2.2, the syntactic notion of traceability is complemented with the semantic notion of equivalence through defining notions of refinement correctness. The semantic integrity is achieved by giving correctness criteria for each category of refinement. The idea behind the correctness criteria is such that, if a criterion holds for a given refinement, then an established property of the original model will hold in the refined model without needing to repeat the verification efforts.

6.2.2 Correctness Criteria

To define the correctness criteria for each category of refinement, the internal details of individual rules must first be syntactically related according to the traceability approach. Listing 6.1 and Listing 6.2 contain two symbolic rules, rule Ri ($1 \leq \mathrm{i} \leq \mathrm{n}$) from model $\mathcal{M}_1$ and rule Sj ($1 \leq \mathrm{j} \leq \mathrm{m}$) from model $\mathcal{M}_2$. Rule Ri contains a time annotation, tr, and p resource annotations, rk. Similarly, rule Sj contains a time annotation, ts, and p resource annotations, sk.

Listing 6.1 Symbolic rule for model $\mathcal{M}_1$

```
Ri: Rule of Model M1
{
  tr  := [rai, rbi];
  r1  := [qi1, ri1];
        ...
  rp  := [qip, rip];

  if RGi then
    REi;
}
```

Listing 6.2 Symbolic rule for model $\mathcal{M}_2$

```
Sj: Rule of Model M2
{
  ts  := [saj, sbj];
  s1  := [uj1, uj1];
        ...
  sp  := [sjp, ujp];

  if SGj then
    SEj;
}
```

The correctness criteria for each refinement type apply to the three aspects of the TASM language - function, time, and resource consumption. The correctness criteria are expressed as relationships between the annotations, the guards, and the effect expressions of the rules contained in the sets of mappings. If certain annotations are not present in a model, there are no restrictions on the behavior of the refined model concerning those annotations. The goal of the correctness criteria is to establish semantic integrity between the two sets of rules, to ensure that semantics are preserved between the two models. For the step expansion, rule expansion, and rule addition refinement types, the idea surrounding the refinement correctness stipulates that if a given semantic property holds in the original model, it will also hold in the refined model if the correctness criteria are met. If the correctness criteria do not hold, no semantic equivalence can be guaranteed by the approach given in this section, although ad-hoc arguments can be used to prove correctness and verification efforts can be exerted to ensure correctness. The correctness criteria do not need to hold for the refinement to be correct, but if they do hold, the amount of verification that needs to be performed on the refined model is reduced because the refined model comes with the semantic guarantees of the original model. If no criteria hold, the refinement comes with no guarantees and verification must be performed on the refined model as if the model is an entirely new model.

The step expansion, rule expansion, and rule addition refinement types capture refinements as introduced through top-down design. The general philosophy surrounding the correctness criteria is that the higher level model dictates the behavior that the refined model must exude. This philosophy relies on the reality that higher level models might typically exist before lower level models. Consequently, higher level models are deemed to be "correct" since analysis can be performed on the high level models before the low level models are developed. The dual of these refinement types, namely the step contraction, the rule contraction, and the rule deletion refinement types, capture refinements as introduced through bottom-up design, with properties that hold in the lower level model being guaranteed to hold in the higher level model, if the correctness criteria hold. For each correctness criterion, the criterion is given, followed by a proof that if the criterion holds, the semantics of the original model are preserved in the refined model.

In this chapter, the term *semantics* is used as a general term to denote a property established in a model using the analysis approaches described in Chapter 5. The idea behind the correctness criteria is such that if the criteria are met for the refinement, properties established in the original model will also hold in the refined model. In the context of the TASM language, the established properties that could hold through the refinement include safety and liveness, expressed as reachable and unreachable states, BCET and WCET, and minimum and maximum amounts of resource consumption. The motivations behind the approach is to reduce the amount of verification that needs to be performed on the refined model. Since traceability is a useful syntactic concept to track and understand design assumptions, adding notions of semantic equivalence to the traceability approach can reduce the duplication of verification activities. The correctness criteria govern the three aspects of the TASM language – function, time, and resources. The correctness criteria are defined for these three aspects by defining conditions that need to hold in the mapping for properties of the original model to hold in the refined model. These conditions govern the three aforementioned aspects:

- Function, through relating the rule guards and rule effects
- Time, through relating the time annotations
- Resources, through relating the resource annotations

In the following subsections, the correctness criteria for each type of refinement is defined in terms of these three aspects. The phrase "semantics are preserved" is used for brevity when stating the theorems to establish the preservation of properties between two models. What is meant by "semantics are preserved" is primarily that safety and liveness analysis performed on the original model is maintained in the refined model. Furthermore, if the conditions relating the time and resource annotations are strict equalities instead of inequalities, and the criteria governing the rule guards and effect expressions hold, WCET and BCET analysis is also preserved. A similar argument can be made for the resource annotations, for the preservation of minimum and maximum resource consumption analysis performed on the original model.

Step Expansion

The correctness criteria for a step expansion refinement is such that the parent rule defines constraints on the set of expanded rules. The step expansion refinement is used to divide the execution of a single rule into two or more consecutive steps. The division is typically achieved by adding an extra variable to the state space, which acts as a program counter to enforce the sequential execution of the refined rules. The refinement of timing behavior requires that the total execution time of refined rules be less than equal to the execution time of the parent. This relationship is logical since the time behavior of the children rules represents a subset of the time behavior of the parent. If verification has been performed on the higher level model, it is logical to assume that established functional properties will still hold in the child model given the subset relationship. Formally, for the rule R_1 shown in Listing 6.1, which is expanded to m rules S_j, one of which is shown in Listing 6.2, the correctness criteria for the time annotations are shown below:

$$\begin{aligned} ra_1 &\leq sa_1 + sa_2 + \ldots + sa_m \\ rb_1 &\geq sb_1 + sb_2 + \ldots + sb_m \end{aligned}$$

Since time annotations are non-deterministic execution times for the rule, verification performed on the model will consider all possible times inside the annotation interval. For the quantitative timing behavior to be preserved through the refinement, such as worst-case execution time, the relationship between the time annotations must be equality instead of inequality. This is necessary since execution time analysis, for WCET and BCET, depends on quantitative values of rule executions and not only on the possible interleavings of rule executions. The correctness criteria for resource consumption follow a similar pattern. However, since resource consumption is additive through parallel steps but not through sequential steps, the relationship involves the maximum and minimum resource consumptions:

$$\begin{aligned} q_{11} &\geq min(s_{11}\ ,\ s_{21},\ \ldots,\ s_{m1}) \\ &\vdots \\ q_{n1} &\geq min(s_{n1}\ ,\ s_{n2},\ \ldots,\ s_{nm}) \\ r_{11} &\leq max(u_{11},\ u_{21},\ \ldots,\ u_{m1}) \\ &\vdots \\ r_{n1} &\leq max(u_{1n},\ u_{2n},\ \ldots,\ u_{mn}) \end{aligned}$$

As for time annotations, for the results of the resource consumption analysis performed on model $\mathcal{M}_1$ to hold in model $\mathcal{M}_2$, the inequalities should be equalities. Furthermore, for resource consumption semantics to hold, the correctness criteria regarding the rule guards must also hold. For the guards, the correctness criteria between $\mathcal{M}_1$ and $\mathcal{M}_2$ is such that the expanded model must not change the semantics of the guard RG_1. Formally, this relationship implies that the disjunction of each guard SG_j must be logically equivalent to RG_1:

$$RG_1 \equiv (SG_1\ \vee\ \ldots\ \vee\ SG_m)$$

Essentially, this relationship states that the disjunction of guards of the expanded rules must form a tautology whenever the guard of rule R_1 is enabled. In other words, on of the expanded rules must execute whenever R_1 would execute. For the effect expressions, the correctness criterion requires that everything that happens in RE_1 must also happen as a result of executing the sequence of refined rules S_j. Per TASM semantics, the effect of executing the rule is applied after the rule execution has completed. Consequently, this effect expression relationship can be expressed as the effect expression of rule R_1 being contained in the intersection of the effect expression of rule R_1 and the union of the effect expressions of the various S_j:

$$RE_1 \cap (SE_1 \cup SE_2 \cup \ldots \cup SE_m) = RE_1$$

Furthermore, the added effects in the rule effect expressions of the refined rules must not affect the execution of any other rule R_i in the model. This restriction also concerns the execution of the original rule R_1. The last condition necessary for the correctness of the refinement requires that the refined rules S_j are executed in an atomic sequence. This means that, for every $1 \leq i < m$, the effect of executing rule S_i causes the guard of rule S_{i+1} to be enabled. Furthermore, no other rule in the model can change the sequence of execution.

These four correctness criteria involving the time, resource consumption, guard, and effect expressions of the two sets of rules form the basis of the correctness criteria for the various types of refinement. If the listed criteria hold, safety, liveness, completeness and consistency results from model $\mathcal{M}_1$ will hold in model $\mathcal{M}_2$. Furthermore, for the execution time analysis to hold, strict equalities must hold for the time annotation criteria. Similarly, for the resource consumption analysis results to hold, strict equalities must hold for the resource annotation criteria. In the following sections, the correctness criteria of other refinement types are explained reusing the notation introduced in this section and the notation from Listing 6.1 and from Listing 6.2. For each type of refinement, the correctness criteria are stated, followed by a argument to prove that, if the criteria holds, the semantics are preserved through the refinement.

Theorem 6.1. *If the correctness criterion for the step expansion refinement holds, the semantics of model $\mathcal{M}_1$ are preserved in model $\mathcal{M}_2$.*

Proof. Suppose that rule R_1 of model $\mathcal{M}_1$ is enabled in state ST_0 and that executing rule R_1 in state ST_0 yields state ST_1. Let $ST_k^{'}$ be the refined state containing ST_k, with the added variable to enforce sequential execution. Since the disjunction of the refined rules S_j form a tautology when rule R_1 is enabled, one of the refined rules must be enabled in state $ST_0^{'}$. The ordering of the execution of refined rules can be selected arbitrarily without loss of generality. In this case, the index of the rules is selected as the order of execution. Let rule S_1 be the enabled rule in state ST_0'. Executing rule S_1 in state ST_0' yields an intermediate state ST_{01}. Since the execution of the refined rules is required to be sequential, the execution of the rules will yield $j-1$ intermediate states. Since the effect of executing rule R_1 must be included in one of the effect expressions of the refined rules, state ST_0 will be contained inside one of the intermediate states and eventually into the final state resulting from executing the refined rules in sequence. From the requirement that no other effect expression can affect any other rule, including rule R_1, it naturally follows that the final state resulting from executing the sequence of refined rules S_j will be ST_0'. □

Step Contraction

The step contraction refinement is the dual of the step expansion refinement, with the direction of the refinement being reversed. Consequently, the correctness criteria detailed in the previous section are exactly the same for this refinement type, but in the reverse direction, including the reversal of the containment relationships. In the rule contraction refinement, the total time of the rules in the refinement for model $\mathcal{M}_1$ specify the constraints on the rule in model $\mathcal{M}_2$:

$$\begin{aligned} ra_1 + ra_2 + \ldots + ra_m &\leq sa_1 \\ rb_1 + rb_2 + \ldots + rb_m &\geq sb_1 \end{aligned}$$

A similar relationship can be defined for the correctness criteria for resource consumptions. This relationship is omitted for brevity. For the relationship between the guards and the effect, the criteria are similar to those detailed for the step expansion refinement:

$$(RG_{11} \vee RG_{12} \vee \ldots \vee RG_{1m}) \equiv SG_1$$
$$RE_1 \cap (RE_1 \cup RE_2 \cup \ldots \cup RE_n) \cap SE_1 = SE_1$$

Theorem 6.2. *If the correctness criterion for the step contraction refinement holds, the semantics of model $\mathcal{M}_2$ are preserved in model $\mathcal{M}_1$.*

Proof. The proof of this theorem is identical to the proof of Theorem 6.1, in the reverse order. $\square$

Rule Expansion

The rule expansion refinement adds information to a rule, in the form of a modified time annotation, modified resource annotations, an expansion of the guard, or an expansion of the effect expression. For time and resource annotations, the correctness criteria require that the annotations in rule R_2 be contained in the annotations of rule R_1. In other words:

$$\begin{aligned} ra1 \leq saj,\ \forall\, j &= 1\ldots m \\ rb1 \geq sbj,\ \forall\, j &= 1\ldots m \\ q1k \leq sjk,\ \forall\, j &= 1\ldots m,\ k = 1\ldots p \\ r1k \geq ujk,\ \forall\, j &= 1\ldots m,\ k = 1\ldots p \end{aligned}$$

The presented relation between the annotations of both models will ensure that functional behavior is maintained for the model, as explained in Theorem 6.3. For execution time analysis and resource consumption analysis to be preserved, the annotations must be equalities instead of inequalities.

If the guard is expanded by adding extra conditions to the guard, the disjunction of the S_j guards in the mapping must be logically equivalent to the guard of R_1.

What this means is that the S_j guards must not be true with variable assignments that make R_1 false and one of the S_j guards must be true whenever R_1 is true:

$$RG_1 \equiv (SG_1 \vee \ \ldots \ \vee SG_m)$$

If the effect expression is expanded, by adding extra assignments of values to variables, the effect expression of R_1 must be contained in the effect expression of S_j:

$$RE_1 \subseteq SE_j, \forall j = 1 \ldots m$$

Furthermore, for any other unchanged rule R_i from the unrefined model, the added items in the refined effect expression must not change the value of the guard of rule R_i. The idea behind the rule expansion refinement is that the behavior of the refined rules are contained within the behavior of the unrefined rule. Consequently, if verification involving the unrefined rule was performed, those results should still hold in the refined rules.

Theorem 6.3. *If the correctness criterion for the rule expansion refinement holds, the semantics of model $\mathcal{M}_1$ are preserved in model $\mathcal{M}_2$.*

Proof. Since the disjunction of the guards of the refined rules are logically equivalent to the guard of the original rule, at least one of the S_j guards of model $\mathcal{M}_2$ will be enabled whenever rule R_1 of model $\mathcal{M}_1$ is enabled. And since the effect expression of rule R_1 of model $\mathcal{M}_1$ is contained in every refined rule of model $\mathcal{M}_2$, the effect of executing rule R_1 is preserved. Furthermore, since the expanded effect expressions do not affect the evaluation of the guards of other rules, the refinement is, in effect, limited to the refined rule and hence preserves the semantics of model $\mathcal{M}_1$. $\square$

Rule Contraction

Rule contraction is analogous to rule expansion and concerns the modification of the time annotation, the modification of resource annotations, the modification of guard conditions, the modification of effect expressions, and the consolidation of rules. The rule contraction refinement would be used when removing variables from the state, resulting in a contraction of the state space and of the number of rules contained in the model. The correctness criteria are similar to the correctness criteria for the rule expansion refinement, but apply in the reverse order:

$$\begin{aligned}
rai \leq sa1,\ \forall\, i &= 1 \ldots n \\
rbi \leq sb1,\ \forall\, i &= 1 \ldots n \\
qik \geq s1k,\ \forall\, i &= 1 \ldots n,\ k = 1 \ldots p \\
rik \leq u1k,\ \forall\, i &= 1 \ldots n,\ k = 1 \ldots p \\
SG_1 \equiv (RG_i \ \vee \ \ldots \ \vee \ RG_n) & \\
SE_1 \subseteq RE_i,\ \forall\, i &= 1 \ldots n
\end{aligned}$$

The idea behind the rule expansion refinement is that the behavior of the unrefined rules is contained inside the behavior of the refined rule. Consequently, if verification involving the refined rule is performed, these results will hold in the unrefined rules.

Theorem 6.4. *If the correctness criterion for the rule contraction refinement holds, the semantics of model $\mathcal{M}_2$ are preserved in model $\mathcal{M}_1$.*

Proof. This proof is identical to the proof of Theorem 6.3. □

Rule Addition

The rule addition refinement refers to a refinement where a set of rules is added to a model. This refinement can be used to add steps in sequential execution or to add parallel rules to handle new conditions resulting from an extended state space. For sets of rules, completeness and consistency have been defined in Section 5.1. The correctness criteria concern the preservation of completeness and consistency through rule addition. The functions *comp* and *cons* are defined between a set of rules and the set {*True*, *False*}. The function returns the value *True* if a set of rules is complete (consistent) and vice-versa [123]. Using the notation from Section 6.2, X_1 is the set of rules of model $\mathcal{M}_1$ and X_2 is the set of rules of model $\mathcal{M}_2$. Using these definitions, the correctness criteria for rule addition ca be defined:

$$\begin{aligned} cons(X_1) &\rightarrow cons(X_2) \\ comp(X_1) &\rightarrow comp(X_2) \end{aligned}$$

What this criteria mean is that adding a rule to an existing rule set must not introduce non-determinism. Non-determinism introduced through the addition of a rule would preserve the semantics only if the effect expressions of the two inconsistent rules would be identical. However, it is not clear why adding a rule that is essentially a copy of another rule would be a useful refinement. If the rule is added to add parallel behavior to handle an extended state space, its effect expression must contain only updates to variables in the extended space. If the rules are added to augment the number of steps in a sequence of execution, updates to the variable representing the order of execution are acceptable, as long as the order of effect expressions is preserved from the original set of rules and do not affect rules outside the machine where the rule is added.

Theorem 6.5. *If the correctness criterion for the rule addition refinement holds, the semantics of model $\mathcal{M}_1$ are preserved in model $\mathcal{M}_2$.*

Proof. The behavior of rules added to reflect the addition of sequential steps is analyzed first. Suppose that m rules S_j are added as intermediate steps between original rule R_1 and R_k. The proof follows the principles of the proof of theorem 6.1. Suppose rule R_1 is enabled in state ST_0 and yields state ST_1, in which state rule R_k is enabled and yields state ST_k after execution. The addition of the rules must follow an atomic sequence where rule must be modified such that executing R_1 must yield a new state ST_1^1 where S_1 is enabled. The added rules must yield an atomic sequence of rule executions that will yield state ST_1 such that executing rule R_k will complete the chain. Since the added rules are required to not have side-effects on other rules, the behavior is preserved.

For rules that are added to handle an extended state space in parallel, the guarantees of completeness and consistency ensure that the rules do not conflict with existing rules in the machine. Furthermore, the requirement that the rules be side-effect free is a restricted case which guarantees that the semantics aren't changed since the rules do not interfere with existing rules. □

Rule Deletion

Similarly, the correctness criteria for rule deletion must not affect the completeness the rule set or the consistency of the rule set where it is added. For example, if a set is complete, removing a rule must not make it incomplete in order to preserve the semantics.

$$
\begin{aligned}
cons(X_1) &\rightarrow cons(X_2) \\
comp(X_1) &\rightarrow comp(X_2)
\end{aligned}
$$

Theorem 6.6. *If the correctness criterion for the rule deletion refinement holds, the semantics of model $\mathcal{M}_2$ are preserved in model $\mathcal{M}_1$.*

Proof. This proof is identical to the proof of Theorem 6.5.

□

Any

The any refinement type does not contain correctness criteria and does not carry guarantees of semantic equivalence. This refinement is used purely for syntactic traceability with no semantic guarantees.

Identity

The correctness criteria for the identity refinement simply ensures that the mapping does not modify the rule:

$$\begin{aligned} tr &= ts \\ s_i &= r_i,\ \forall\ i\ =\ 1\ \ldots\ m\ ,\ m\ =\ n \\ RG_1 &= SG_1 \\ SE_1 &= RE_1 \end{aligned}$$

It is straightforward from this identity mapping to verify that the mapping does not modify the semantics of the model.

6.3 Example

In this section, an example is provided to illustrate the concepts explained throughout this chapter. In Chapter 8, the Electronic Throttle Controller (ETC) case study provides a more complex example of bi-directional traceability between disparate models, with the correctness criteria ensuring semantic guarantees between models. The refinements and the traceability properties of the ETC case study are presented in Section 8.4. In this section, the light switch example, originally presented in Section 4.1.5, is refined, first by expanding the state space and secondly through the addition of tasks and a scheduler. The light switch example contains two components, a light bulb and a switch. The controller software is responsible for turning on the light based on the switch position. A high level model might contain simple logic to describe the behavior of the software, as shown in Listing 6.3. The model of Listing 6.3 describes the requirement "if the switch is UP, the light shall be ON" and the requirement "if the switch is DOWN, the light shall be OFF".

Listing 6.3 Model 1 of light switch

```
R1: Turn on
{
  if switch = UP then
    light := ON;
}

R2: Turn off
{
  if switch = DOWN then
    light := OFF;
}
```

A refinement of the requirements might extend the functionality of the light by introducing additional conditions on the behavior of the software, leading to a refined model. For example, Listing 6.4 describes behavior that meets the same requirement as in Listing 6.3 but incorporates the extra requirement "when turning the light ON during day time, the light intensity shall be LOW while it shall be HIGH during nighttime". This extra functionality is added to the model of Listing 6.3 by expanding the state space. Two new user-defined types are added – one to describe the time of day (values "DAY" and "NIGHT") and one to describe the intensity of the light (values "LOW" and "HIGH"). Two extra variables are added to the state to denote the time of day and the light intensity.

Establishing syntactical traceability between model 1 and model 2 is fairly straightforward. The refinement type for the mapping between the two rules is of the "Rule Expansion" type, where rule R_1 is mapped to rules S_{11} and S_{12}. In the refinement, items are added to the rule guards and to the effect expressions. Formally, the syntactical mapping is:

Listing 6.4 Model 2 of light switch

```
S11: Turn on, low {
  if switch = UP and timeofday = DAY then
    light     := ON;
    intensity := LOW;
}

S12: Turn on, high {
  if switch = UP and timeofday = NIGHT then
    light     := ON;
    intensity := HIGH;
}

S2: Turn off
{
  if switch = DOWN then
    light := OFF;
}
```

$$\begin{aligned} \mathrm{T} = \mathbb{T}_{rexp} \cup \mathbb{T}_{id} &= (\langle\{R_1\}, \{S_{11}, S_{12}\}\rangle) \cup (\langle\{R_2\}, \{S_2\}\rangle) \\ &= (\langle\{R_1\}, \{S_{11}, S_{12}\}\rangle, \langle\{R_2\}, \{S_2\}\rangle) \end{aligned}$$

Since the guards of the rules are changed in model 2, the correctness criteria for the rule expansion refinement must establish guard equivalence. The equivalence can be visualized in Table 6.1 for rule R_1. The truth table clearly shows that the disjunction of rule S_{11} and rule S_{12} is equivalent to the guard of rule R_1.

`switch = UP`	`timeofday = DAY`	`timeofday = NIGHT`	R_1	S_{11}	S_{12}	$S_{11} \vee S_{12}$
T	T	F	**T**	T	F	**T**
T	F	T	**T**	F	T	**T**
F	T	F	**F**	F	F	**F**
F	F	T	**F**	F	F	**F**

Table 6.1: Truth table to verify the correctness criteria for the rule expansion refinement between rule R_1 and rules S_{11} and S_{12}

Since the model contains only one machine, no time annotations, and no resource annotations, it is clear that the correctness criteria hold for the rule expansion refinement. Furthermore, the changes in the effect expressions of the refined rules do not affect any other rules, preserving the semantics of model 1. What this means is that a statement made about the possible traces of the model 1 will hold in the refined model. For example, the property that a "state where the switch is UP will eventually always be followed by a state where the light is ON" will hold in both models.

In order to add interesting features to the model, a scheduler driven by a 1 ms clock is added to the model. The scheduler fires a task that has a period of 30 ms. The task has an execution time lasting between 1 and 10 ms. The scheduler is added as an extra main machine, in order to drive the system and switch the task's status between "wait" and "exec". The task is another main machine which simply waits to be activated. The scheduler is shown in Listing 6.5 and the task is shown in Listing 6.6.

Listing 6.5 Model of the scheduler with 1 ms clock, firing a task with a period of 30 ms

```
T1: Fire
{
  t := 1;

  if tick = 30 then
    tick := 1;
    task := exec;
}

T2: Tick
{
  t := 1;

  else then
    tick := tick + 1;
}
```

Listing 6.6 Model of the task, model 3

```
P1: Execute
{
  t := [1, 10];

  if task = exec then
    task := wait;
}

P2: Wait
{
  t := next;

  else then
    skip;
}
```

In Listing 6.4, the task is simply a placeholder which does not provide any functionality other than consume time. The model of the task can be combined with model 2 of Listing 6.4 to wrap the functionality of model 2 inside of the tasking model. The resulting model is shown in Listing 6.7.

Listing 6.7 Combined model with task implementation, model 4

```
D1: Turn on, low
{
  if task = exec and switch = UP and timeofday = DAY then
    light     := ON;
    intensity := LOW;
    switch    := wait;
}

D2: Turn on, high
{
  if task = exec and switch = UP and timeofday = NIGHT then
    light     := ON;
    intensity := HIGH;
    task      := wait;
}

D3: Turn off
{
  if task = exec and switch = DOWN then
    light := OFF;
}

D4:
{
  t := next;

  else then
    skip;
}
```

Because model 4 is a refinement of model 3 and a refinement of model 2, the traceability relationship yields two branches. The branch between model 3 and model 4 yields the following syntactic traceability relationship:

$$\mathrm{T} = \mathbb{T}_{rexp} \cup \mathbb{T}_{id} = (\langle\{P_1\}, \{D_1, D_2\}\rangle) \cup (\langle\{P_2\}, \{D_4\}\rangle)$$
$$= (\langle\{P_1\}, \{D_1, D_2\}\rangle, \langle\{P_2\}, \{D_4\}\rangle)$$

The correctness criteria for the rule expansion refinement between rule P_1 and rules D_1, D_2, and D_3 is shown in Table 6.2. Since the variables involved in the refinement are binary variables, there is no need to include columns for the value "NIGHT" and for the value "wait", since these values are captured in the columns for the value "DAY" and for the value "exec". It is clear from the table that the correctness criteria hold for the guards of the refined rule. The time and resource annotations also conform to the correctness criteria. The mapping between rule P_2 and rule D_4 is trivially correct because the two rules are exactly the same since it is an identity refinement.

`switch = UP`	`timeofday = DAY`	`task = exec`	P_1	D_1	D_2	D_3	$D_1 \vee D_2 \vee D_3$
T	T	T	**T**	T	F	F	**T**
F	F	F	**F**	F	F	F	**F**
T	F	F	**F**	F	F	F	**F**
F	T	F	**F**	F	F	F	**F**
F	F	T	**T**	F	F	T	**T**
T	T	F	**F**	F	F	F	**F**
T	F	T	**T**	T	F	F	**T**
F	T	T	**T**	F	F	T	**T**

Table 6.2: Truth table to verify correctness the criteria for the rule expansion refinement between rule P_1 and rules D_1, D_2, and D_3

Furthermore, since the added effect expressions from model 3 to model 4 concern only the newly introduced variables, the locality condition of the correctness criteria also holds. It is interesting to note that the refinement from model 3 to model 4 could have been achieved through hierarchical composition, by wrapping the functionality of model 2 inside a submachine. In this case, the correctness criteria with regards to the guards would require the sub machine to be complete, leading to the appropriate correctness criterion for the guards, according to Theorem 5.1. The machine depicted in Listing 6.7 corresponds to the "flattened" machine that would be obtained by removing the hierarchical composition, as explained in the proof of Theorem 4.2. Since the correctness criteria hold for the refinement from model 3 to model 4, the properties of model 3 are preserved in model 4. For example, the worst-case execution time of each task is preserved, as are the schedulability attributes. A similar approach is used in Section 8.4 for the ETC case study, where the functionality of the controller is implemented using a set of tasks.

Model 4 can also be viewed as a refinement of model 2. The traceability branch between model 2 and model 4 yields the following syntactic traceability relationship:

$$\begin{aligned} \mathrm{T} = \mathbb{T}_{rexp} \cup \mathbb{T}_{add} &= (\langle\{S_{11}\},\ \{D_1\}\rangle,\ \langle\{S_{12}\},\ \{D_2\}\rangle,\ \langle\{S_2\},\ \{D_3\}\rangle) \quad \cup \\ &\quad (\langle\{\},\ \{D_4\}\rangle,\ \langle\{\},\ \{T_1, T_2\}\rangle) \\ &= (\langle\{S_{11}\},\ \{D_1\}\rangle,\ \langle\{S_{12}\},\ \{D_2\}\rangle,\ \langle\{S_2\},\ \{D_3\}\rangle, \\ &\quad \langle\{\{\},\ D_4\}\rangle,\ \langle\{\},\ \{T_1, T_2\}\rangle) \end{aligned}$$

The correctness criteria for the mapping between rule S_2 and rule D_3 resembles the criteria for the mapping displayed in Table 6.2. To be more thorough and for the correctness criteria to hold, extra rules should be added to model 4 to reflect the cases where the task is in the "wait" state. However, this mapping is guaranteed by the "Else" rule. The mapping preserves the semantics of model 2 since the modifications are local to the expanded state space. A similar argument can be made for the rule expansion refinements of rule S_{11} and rule S_{12}. The correctness criteria get a bit more complex to visualize for the rule addition refinements since the addition of the "Else" rule is done to modify the termination semantics of the machine and to handle the case where the task is in the "wait" state. Such a dual purpose refinement makes the model simpler, but exacerbates the proof of the preservation of semantics. In order to make the preservation easier to demonstrate, model 2 could have already contained the "Else" rule and model 4 could have contained 3 extra rules to handle the case where the task is in the "wait" state. Furthermore, the addition of the scheduler, which acts on the refined rules, complicates the problem. Nevertheless, using simple arguments, it is possible to see that the addition of the scheduler does not affect the order of execution of the functional model, and the property "if the switch is UP, the light will always eventually be ON", from model 1 still holds. Properties from model 2 with regards to the light intensity also hold.

This simple example provides an illustration of the concepts outlined in this chapter. The example provides an interesting application of the proposed approach, to combine a tasking model and a functional model, all the while preserving some behavioral semantics during the refinement process. In the first branch of traceability, scheduling attributes were preserved while in the second branch, functional attributes were preserved. Section 8.4.2 uses a similar technique to achieve traceability and refinement correctness on a more complex example using the Electronic Throttle Controller (ETC) case study.

6.4 Segue into Chapter 7

This chapter presented an approach to relate different TASM models syntactically, through a mapping between the rules of the two models. The proposed strategy also presented a set of archetypical refinement types, as surveyed through literature and experience with modeling. Each refinement type is accompanied by a set of correctness criteria which, if satisfied, preserve certain semantic aspects between the two models. The approach was demonstrated using an extended version of the light switch example from Section 4.1.5. The following chapter, Chapter 7, presents an approach to generate test cases automatically based on TASM specifications. The presented method enables the automated generation of unit test cases, integration test cases, and regression test cases.

Chapter 7

Test Case Generation

This chapter presents an approach to automatically generate test cases based on a model expressed in the TASM language. The generation of test cases is achieved for unit testing, integration testing, and regression testing. The test cases are generated to achieve coverage of the TASM model according to the rule coverage criteria, as defined in the ASM community. Unit test cases are generated in the context of an individual machine without hierarchical composition. Integration test cases are generated by combining unit test cases hierarchically. The traceability relationship, described in Chapter 6, is leveraged to provide an approach to the generation of regression test cases. The generation of unit, integration, and regression test cases is provided in separate sections and illustrative examples are given at the end of each section.

7.1 Related Work

Testing remains the main activity in industry and in software engineering circles to build confidence into the system being engineered [220, 242]. Even though testing can never establish the absence of defects [81], the popularity of testing has grown the practice and theory considerably, leading to well-established definitions and concepts in the software engineering community [26, 169, 176].

The growing popularity of testing has led to various approaches to automatically generate test cases [74, 86] and to the development of various tools [27]. More recently, the advent of model-based software engineering has given birth to model-based testing, an approach to test case generation where a model or specification is used as the basis to generate test cases [32, 130, 222]. Model-based testing builds upon previous results from the requirements engineering community where requirement specifications are used to generate test cases [75, 112, 245]. In model-based testing, two key approaches are used to automatically generate test cases, constraint-based test case generation [214] and test case generation using model checkers [8]. Test case generation using model checkers relies on some form of automata model to generate *test sequences* that are used to *cover* certain aspects of the model such as states and transitions [102]. Notably, the UPPAAL tool suite has been used to generate test cases for real-time systems in [128]. In [128], synchronization channels are used to model the inputs and outputs of the system and the automata model is assumed to be "input enabled and output urgent" to ensure that the generated test suite is time optimal [36]. The approach described in [128] is not applicable to TASM specifications because the language does not contain synchronization channels. Furthermore, the generation of test sequences, as performed using model checkers, relies on assumptions about the system under test, which may not be applicable in practice, as explained in Section 7.5.1. The approach presented in this chapter does not preclude the generation of test sequences, but provides a more generic approach to test case generation which can be tailored to a specific purpose.

The test case generation strategy presented in this chapter relies on constraint programming and symbolic combination of test cases, a strategy related to the approach presented in [15] and in [16]. However, the definition of *templates* is unique to the TASM-based approach although it resembles the theory of equivalence classes explained in [176]. The approach presented in this chapter uses the specification as an *oracle*, that is, as the authority on the *correct* input/output behavior of the system under test [219]. In other words, the specification describes the expected output of the system for each class of inputs. Much of the theory on test case generation concerns unit testing; the strategy described in this chapter extends unit testing capabilities by combining unit test cases to achieve integration testing. Furthermore, the approach presented in this chapter uses traceability attributes to generate regression test cases in the occurrence of specification modifications or extensions. The regression test case generation uses structured model changes as the basis for test case generation, similar to mutation-based approaches were a well-defined set of defects are used to generate test cases [247]. This section reviewed the popular approaches to test case generation at a high level and situated the proposed approach in light of past efforts. In the following subsections, different coverage criteria are reviewed and test case generation efforts within the Abstract State Machine (ASM) community are reviewed.

7.1.1 Coverage Criteria

The purpose of performing testing activities is to achieve a desired level of confidence into the functionality of the system being tested. However, in order to achieve a satisfactory level of testing, some criteria need to be put in place to decide what to test and when to stop the testing activities. For this reason, traditional ad-hoc approaches to testing have been replaced with structured approaches to testing whose aim is to achieve a predetermined level of coverage of the system under test [176]. The possible levels of coverage are captured into coverage criteria, which express properties of the system under test, such as program branch coverage and program variable definition and usage coverage. The list of possible coverage criteria is quite large and a good survey is provided in [256]. In the field of safety-critical systems, the DO-178B standard requires that an implemented system of a certain criticality level be tested to achieve the Modified Condition/Decision Coverage (MC/DC) criterion [64, 216]. Standard DO-178B also requires that requirements be tested, expressing that the testing of the implemented system also cover all requirements. In the field of model-based testing and specification-based testing, the coverage criteria used for generating test cases refer to coverage items of the model such as states and transitions [153, 184].

In the context of Abstract State Machines (ASM), coverage criteria have been defined in [103]. The coverage criterion used in the approach presented in this chapter is the *rule coverage criterion* presented in [103]. The rule coverage criterion is analogous to the transition coverage criterion described in [184] and in [256] which concern variants of automata.

7.1.2 Abstract State Machines

In the ASM community, test cases have been generated automatically based on specifications, using model checking techniques [104]. The approach generates test sequences to achieve different coverage criteria, including rule coverage. The approach presented in this chapter differs from the approach presented in [104] in that it uses constraint logic programming to generate test cases. Furthermore, the approach presented in this chapter also generates test cases and not test sequences. Another approach in the ASM community generates test cases based on the AsmL language, an ASM derivative developed at Microsoft [116]. The test case generation approach derives a finite state machine approximation of an AsmL specification for the purpose of generating test cases using established algorithms [110]. The test case strategy is accompanied by a tool for automated generation [109]. The AsmL approach to test case generation provides an underapproximation of the true finite state machine, as explained in [110] through the execution of the specification. The approach presented in this chapter differs from the AsmL approach in that it does not rely on generating a finite state machine. However, the concept of *hyperstate* described in [110] is closely related to the concept of *test case template* described in this chapter. A benefit of the approach described in this chapter is the ability to generate test cases incrementally and locally, and combining the test cases to achieve coverage of the specification. Furthermore, the generation of regression test cases has not been addressed in the ASM community.

7.2 Test Case Generation Concepts

This section provides definitions of concepts that are used in the three subsequent sections when describing the test case generation algorithms. The provided definitions are defined by the presented research. However, many of the concepts have already been defined in test case generation theory [176]. Where applicable definitions borrowed from the testing community are cited. The concept of a *template* is unique to the test case generation approach presented in this chapter although the notion of template has been used extensively in other software engineering branches.

7.2.1 Definitions

The test case generation strategy uses concepts from set theory to generate test cases symbolically. Throughout the description of the algorithms, the terms *template* and *instance* are used to explain the approach. At a high level, a *template* is a generic concept which describes a family or a set of items which share a common property. This definition is analogous to the definition of a set in discrete mathematics [232]. Analogously, an *instance* is a single member of a template, analogous to an *element* of a set in discrete mathematics. The terms *template* and *instance* are used over *set* and *member* because the terms are well-established terms in the software engineering community. Furthermore, for a template/instance pair, the term *template* is used to describe the properties of the members of the template while the term *instance* will be generally omitted for brevity. For example, in the rest of this section terms like *variable template* and *test case template* are used to denote a set of items sharing a given property while the terms *variable* and *test case* are used to denote instances of the templates.

A *variable template* is a variable name accompanied by a set of possible values for the variable. A variable template is analogous to a datatype but the term template is used to maintain consistent naming across concepts. For example, a variable template for an integer variable named a with a lower bound of -11 and an upper bound of 50 would be defined as "$\texttt{a}\{\texttt{-11} \leq \texttt{a} \leq \texttt{50}\}$". A *variable instance* or variable for short, is the variable with an associated value taken from the template, which, in this case, is the interval `[-11, 50]`. For example, the variable "`a = 0`" is an instance of the variable template "$\texttt{a}\{\texttt{-11} \leq \texttt{a} \leq \texttt{50}\}$". Analogously, a *state template* is a set of variable templates. A *state instance*, or *state* for short, is a member of a state template where each member of the state are variables which are, in turn, instances of the variable templates defined in the state template. The definition of a *test case instance* is given before the definition of a *test case template* to avoid crowding the definitions with the word *template.*

A *test case instance* or *test case* for short, TC, is a pair of state instances $\langle S, S' \rangle$ where S is the *pre state* and S' is the *post state.* A test case describes expected behavior of a model as follows: "if a step of the model is executed in state S, the resulting state will be state S'". It is important to note that the state S does not need to be a *complete state*, that is, a state containing values for all the variables of the model. The state S should contain enough variable values to exercise an aspect of interest of the model, for example, a specific rule. Similarly, the state S' also does not need to be a complete state, but needs to contain the expected changes in the state caused by the test case. Item of the state not included in the pre state or in the post state are assumed to have no effect on the purpose and outcome of the test case.

A *test case template*, TCT, is a family or set of test cases which exercise the same aspect of the model. A test case template is composed of a pre state template, ST, and of a post state template, ST'. For example, a test case template might dictate that the pre state "`a > 3, b < 10`" exercises a desired aspect of the model with expected post-state "`a = a + b`". The provided test case template describes a set of test cases containing numerous, potentially infinite, possibilities for test cases that satisfy the template. A test case is an *instance* of a test case template, whose pre state and post state are contained in the pre state template and in the post state template of the test case template. For example, the test case $\langle$ "`a = 5, b = 6`", "`a = 11`" $\rangle$ is an instance of the test case template $\langle$ "`a > 3, b < 10`", "`a = a + b`" $\rangle$.

If some form of model coverage is the goal of the test case generation, the test case and the test case template concepts can be augmented to include the intended coverage item for the test case or test case template. A *coverage test case* (alternatively, *coverage test case template*) is a pair that relates a test case TC (alternatively, test case template TCT) with a coverage item CI: $\langle TC, CI \rangle$ (alternatively, $\langle TCT, CI \rangle$).

A *test suite instance* or *test suite* for short, TS, is a set of coverage test cases which achieves a specific purpose. A *coverage test suite* is a set of coverage test cases which seeks to achieve a specific level of coverage. A coverage test suite is said to be *adequate* if its set of taste cases collectively achieve a desired coverage criterion of the model. A coverage test suite is said to be *minimal* if its set of test cases is adequate and no two test cases exercise the same coverage item. The definitions of adequate and minimal are congruent with established test case generation theory [176].

In these definitions, the assumption is such that the model can be started in any state and that the state is fully observable. The post state is observed after executing a single step of the model. While this assumption might not be immediately adequate in practice, it provides greater flexibility when generating the test cases. As explained in Section 7.5, the test case generation strategy is generic and can be adapted to the specific properties of the system being tested.

In the presented framework and in this chapter, the coverage criterion used for test case generation is the "rule coverage criterion", which requires that all rules of the specification be exercised [103]. Given this definition, an adequate test suite for a given TASM specification would be a set of test cases which collectively exercise all rules in all machines of a TASM specification. Given the rule coverage criterion, in the context of TASM, a coverage test case CTC_i would be of the form $\langle \langle S, S' \rangle, \{ R_i \} \rangle$, where "$\{ R_i \}$" denotes a set of rules covered by the test case. For a TASM specification $\mathcal{M}$ with n rules $\{S_1, \ldots, S_i, \ldots, S_n\}$, for an adequate coverage test suite for specification $\mathcal{M}$, the union of the coverage items in each coverage test case contained in the test suite must be equal to the set of rules in the TASM specification $\mathcal{M}$.

7.2.2 Operations on Templates

In Section 7.4, test case templates generated for unit testing are combined to derive integration test case templates. In doing so, operations are performed on test case templates. Because the templates are defined hierarchically, test case templates contain state templates, which, in turn, contain variable templates. To perform the combination of templates, operations from set theory are used, including intersection ($\cap$) and union ($\cup$). The combination operations are distributed inward and are applied in a manner congruent with traditional set theory. For example, taking the intersection of two test case templates would involve taking the intersection of the two pre states. Taking the intersection of the two pre states involves taking the intersection of the two sets of variable templates. When taking the intersection of the variable templates, the result of the intersection contains only the variables common to both sets. Furthermore, the definition of each variable template would be intersected. For template definitions defined as enumerations, the intersection is applied using traditional set theory. For template definitions defined as inequalities, the intersection is applied according to the principles of interval arithmetic.

In its basic form, interval arithmetic is the definition of operations where the operands are intervals. In the test case strategy, interval arithmetic is useful in two facets – when combining test case templates and when computing the post state template based on a pre state template. Since variable templates for integer and for real variables can be expressed as equalities using $>$, $<$, $\leq$, and $\geq$, calculating the post state template involves performing operations on templates. For templates that contain sets of values for a variable, normal set operations apply and arithmetic operations are defined as the cartesian product of operations. If a variable is "free" in a test case template, that is, it appears only as a right-value, bounds on the values of the variable can be provided by the user or can be obtained from the variable definition in the TASM specification. The following subsection describes the principles of interval arithmetic used in the test case generation strategy.

Interval Arithmetic

The operations performed on integer-valued and real-valued variables expressed as intervals follow the basic conventions of interval arithmetic, as described in [148] and in [145]. These operations are summarized below, following the convention that "[" and "]" are used to express inclusion for a bound of the interval, while "(" and ")" are used to express exclusion for the associated bound of the interval.

Pre State	Post State
$number\{10\}$, $loaded\_blocks\{[LB, 9), [20, UB]\}$, $feed\_belt\{empty\}$	$loaded\_blocks\{[LB+1, 10), [21, UB+1]\}$ $feed\_begin\{True\}$, $feed\_belt\{loaded\}$

Table 7.1: Pre and post state for sample test case template for Listing 7.1

$$\begin{array}{llcll}
[a,b] & + & [c,d] & = & [a+c, b+d] \\
[a,b] & - & [c,d] & = & [a-d, b-c] \\
[a,b] & * & [c,d] & = & [min(ac,ad,bc,bd), max(ac,ad,bc,bd)] \\
[a,b] & / & [c,d] & = & [min(a/c,a/d,b/c,b/d), max(a/c,a/d,b/c,b/d)] \\
[a,b] & \cap & [c,d] & = & [max(a,c), min(b,d)] \\
[a,b] & \cup & [c,d] & = & [min(a,c), max(b,d)] \\
\{(,)\} & op & \{[,]\} & = & \{(,)\}
\end{array}$$

The last operation denotes that an operation between an inclusive bound and an exclusive bound results in an exclusive bound. It is important to note that set operations and interval arithmetic can be combined. For example, for the rule given in Listing 7.1, the relevant test case template and test case instance involves set operations and interval arithmetic. In the rest of this chapter, the notation "varnameval$_1$, …, val$_n$" is used to denote that variable "varname" can take on any value "val$_i$", where "var$_i$ can be an interval. The example from Listing 7.1 yields the test case template described in Table 7.1. In the table, the interval bounds LB and UB denote the upper bound and lower bound values of the variable, as specified in the TASM environment definition. Since the variable **number** is free in the test case, its value was arbitrarily selected to be 10, although an interval for the variable could also have been derived from the variable definition in the TASM specification.

The test case template given in Table 7.1 can be easily converted to a coverage test case template by adding the R_1 rule to the pre state and post state pair. Furthermore, since a template describes a family of test cases, it can be instantiated to yield a coverage test case. An instance of the coverage test case template is shown in

Listing 7.1 Rule for sample test case template

```
R1: The first rule
{

  if (loaded_blocks < number - 1 or loaded_blocks >= 20) and feed_belt = empty then
    feed_belt       := loaded;
    loaded_blocks   := loaded_blocks + 1;
    feed_begin      := True;
}
```

Pre State	Post State	Coverage Item
$number\{10\}$, $loaded\_blocks = 8$, $feed\_belt = empty$	$loaded\_blocks = 11$, $feed\_begin = True$, $feed\_belt = loaded$	R_1

Table 7.2: Coverage test case corresponding to the template of Table 7.1

Table 7.2. It is fairly straightforward to see that the test case shown in Table 7.2 is, indeed, an instance of the test case template shown in Table 7.1.

7.2.3 Machines and Test Suites

The strategy presented in this chapter uses a combination of unit test case generation and integration test case generation to generate test suites to cover all the rules of all machines in a TASM specification. In the test case generation strategy, a generated test suite is associated with a given machine in the TASM specification. In the following sections, the terminology "the test suite of machine M" is used to describe the test suite that covers the rules of machine M. Regardless of whether the test suite is generated for unit testing or for integration testing, the generated test suite is derived to cover a specific machine and remains associated with that machine. The association can be visualized as a pair between a machine and a test suite $\langle \mathrm{M}, TS \rangle$. Of course this relationship also holds for other variations of test suites such as test suite templates and coverage test suites.

7.3 Unit Test Case Generation

Unit testing refers to the testing of a piece of software in isolation. In programming language terms, unit testing could concern the testing of a function, the testing of a class, or the testing of an algorithm [176]. The term *unit* is used to denote that a small piece of the total program (in this case a small piece of the specification) is targeted and that other pieces of the program are abstracted away in the test case generation and execution. In terms of a TASM specification, the basic units of structural organization are rules and machines. While the rules are used as the coverage items, the machines are used as the basic units for test case generation. In the unit test case generation strategy, it is assumed that the machine for which test cases are being generated is "flat", meaning that it does not contain hierarchical composition. Per Theorem 4.1 and Theorem 4.2, any machine with hierarchical composition could be "flattened". However, the test case generation strategy does not necessarily require that a machine with hierarchical composition be flattened before generating test cases. While this approach would work using the unit test case generation algorithm presented in this section, Section 7.4 considers hierarchical composition as part of the integration test case generation strategy; treating hierarchical composition as part of the integration testing strategy leads to better complexity results than flattening the machine and using the unit test case generation algorithm. Any TASM model that contains hierarchical composition will also contain at least one machine which does not contain hierarchical composition. The complete test case generation algorithm described in Section 7.5 explains how the unit test case generation algorithm is used on machines that do not use hierarchical composition and how the integration test case generation algorithm combines the unit test cases for machines that use hierarchical composition.

Since the algorithm performs the calculation of the pre state template and of the post state template for each rule, the generated test suite template will be adequate if, for each rule, the guard of the rule is satisfiable. Furthermore, if the machine is consistent, the test suite will be minimal. The unit test case algorithm can be

Listing 7.2 Unit test case generation algorithm for a machine M

```
• For each rule $R_i$ of machine M:
    – Create pre state template $PreS_i$:
        ∗ Bind each free variable in the guard $G_i$
        ∗ For each other variable $v_{ij}$ in the guard $G_i$
            · Find sets of values that make $G_i$ True
    – Create post state template $PostS_i$:
        ∗ Calculate the post state template by executing $R_i$ on the pre state
          template
    – Create test case template $TCT_i$:
        ∗ $TCT_i = \langle \langle PreS_i, PostS_i \rangle, R_i \rangle$
        ∗ Add the test case template $TCT_i$ to the template test suite $TCTS$
• Associate the test suite $TCTS$ with machine M: $\langle$ M, $TCTS$ $\rangle$
```

implemented by reusing the mapping to *SAT* described in Appendix B, for TASM specifications whose rule guards meet the characteristics described in the appendix. The SAT4J *SAT* solver provides the set of all solutions that satisfy the propositional formula given as the *SAT* instance. Given the "discretization" of the state involving integer and real variables, as explained in Appendix B, the iteration of solutions can easily be aggregated to yield the pre state template and the post state template can be calculated using the operations described in Section 7.2.

7.3.1 Complexity Analysis

Because the algorithm described in Listing 7.2 operates on a machine that does not contain hierarchical composition, the generated number of test case templates generated will be equal to the number of rules for machine M. The number of generated test case templates will always be fixed for the algorithm in Listing 7.2. However, the properties of the guard of each rule greatly affects the complexity of generating the test case. For an implementation using a *SAT* solver, the complexity analysis for the translation is available in Section B.3.1. However, contrary to the usage of the *SAT* solver for the verification of completeness and consistency where demonstrating the existence of a solution is the goal, the test case generation algorithm needs to iterate through all solutions and aggregate the results into the pre state template. The performance of the aggregation will be linear in the number of solutions since each variable can be expanded as needed depending on the properties of the solution. For generating the post state template, simple arithmetic is necessary, leading to linear performance in the number of variables. The complexity of test case generation resides in the translation of the rule guards to *SAT* and executing the resulting problem through the solver. The complexity of the test case generation itself is fairly straightforward.

7.3.2 Example

A short example is provided to illustrate the algorithm described in Listing 7.2. The example provides 3 rules of a sample machine from the Timeliner case study, with some modifications to illustrate the concepts of interval arithmetic and test case templates. The Timeliner case study is analyzed in details in Section 8.5. In the context of the example, the meaning of the machine is irrelevant as it is used solely to generate test cases. In Listing 7.3, the variable NOMINAL_TEMP_MID is a constant equal to 25 and the lower bound of the temperature variable is -10 and its upper bound is 40, inclusively.

Listing 7.3 Rules of the SEQUENCE_TEMP_MONITOR_WORK sub machine (partial)

```
R7: b3 -> b4 {
  t := 2390;

  if temp_seq_b = b3 and temperature <= 19 then
    temp_seq_b  := b4;
    heating     := on;
    delta       := NOMINAL_TEMP_MID - temperature;
}

R8: b4 -> b4 {
  t := 1630;

  if temp_seq_b = b4 and temperature < 22 then
    temp_seq_b  := b4;
    temp_seq_s  := done;
}

R9: b4 -> b0 {
  t := 3195;

  if temp_seq_b = b4 and temperature >= 22 then
    temp_seq_b  := b0;
    heating     := off;
    temp_seq_s  := done;
}
```

The generation of unit test cases for the machine of Listing 7.3, yields 3 coverage test case templates, one to cover each rule of the machine. The 3 test case templates are listed in Table 7.3. The calculation of the post state template for rule R_7 utilizes the interval arithmetic rules from Section 7.2 for subtraction. While this example is

Pre State	Post State	Coverage Item
$temp\_seq\_b\{b3\}$, $temperature = \{[-10, 19]\}$	$temp\_seq\_b\{b4\}$, $heating\{on\}$, $temperature\{[6, 35]\}$	R_7
$temp\_seq\_b\{b4\}$, $temperature = \{[-10, 22)\}$	$temp\_seq\_b\{b4\}$, $temp\_seq\_s\{done\}$	R_8
$temp\_seq\_b\{b4\}$, $temperature = \{[22, 40]\}$	$temp\_seq\_b\{b0\}$, $heating\{off\}$, $temp\_seq\_s\{done\}$	R_9

Table 7.3: Template test suite for the machine of Listing 7.3

somewhat simple, it is taken almost verbatim from the case study. This example will also be reused in Section 7.4 and in Section 7.6 to illustrate the integration test case generation strategy and the regression test case strategy.

7.4 Integration Test Case Generation

Integration testing concerns testing the combination of two or more units, eventually resulting in the complete system being exercised. In the context of TASM specifications, the combination of units occur during hierarchical composition, where one unit uses another unit in an effect expression either as a sub machine or as a function machine.

7.4.1 Hierarchical Composition

In the TASM language, hierarchical composition is achieved via function machines and sub machines. According to Theorem 4.1 and Theorem 4.2, hierarchical composition can be removed, yielding an equivalent "flattened" machine without hierarchical composition. Consequently, the algorithm for generating unit test cases could be applied to the equivalent "flattened" machine, removing the need for a special algorithm to generate integration test cases. However, there are two main reasons justifying the need for an algorithm for integration testing. First, unit testing typically happens before integration testing, meaning that the set of unit test cases for different machines will have already been generated when integration testing begins.

Consequently, reusing the unit test cases would save a certain amount of work since the test case generation strategy does not need to start fresh. Secondly, obtaining the equivalent "flattened" machine through the approach explained in Theorem 4.1 and in Theorem 4.2 can lead to exponential growth in the number of rules of the "flattened" machine, if multiple units of hierarchical composition are used within a rule. The integration test case generation described in this section eliminates the need to flatten the machine, hereby avoiding possible exponential growth in the numbers of rules.

To make the generation of integration test cases slightly simpler, the approach assumes that there are no function machines used in the rule guards. This assumption is valid since the rule guards could easily be rewritten without the use of function machines, as explained in the proof of Theorem 4.1. Furthermore, another simplification involving function machines is used to ease the generation of integration test cases. A function machine can be converted to an equivalent sub machine by converting the arguments to the function machine into fresh environment variables, which become monitored variables of the sub machine. Furthermore, the variable to which the return value of the function machine is assigned can be added as a controlled variable of the sub machine. In the function machine, the output variable is replaced by the controlled variable. Given these two simplifications, the test case generation algorithm can be expressed as the combination of unit test cases of sub machines, as explained in Listing 7.4. Suppose that a given machine M uses a sub machine SM in the effect expression of one of its rules, rule R_i. Also suppose that a coverage test suite template has been generated for submachine SM using the algorithm described in Section 7.3. Machine M and sub machine SM could share monitored variables. If they do, the algorithm must take into account the possibility that the rules of the two machines could be enabled under different conditions for the same variables. It is assumed that the machines do not share controlled variables because this would result in update set inconsistency per ASM theory [42]. In the description of the algorithm, it is assumed that the coverage test suite template for machine SM contains m test case templates, of the form $TCT_{SM,j} = \langle \langle \mathit{PreS}_{SM,j}, \mathit{PostS}_{SM,j} \rangle, S_j \rangle$, where the subscript SM, j is used to denote the machine to which the test case belongs, in this case machine SM and the rule covered by the test case, in this case the j^{th} rule effect expression.

In the algorithm of Listing 7.4, if the two machines do not share monitored variables, the resulting test suite is simply the union of the test case templates for machine SM with the test case template for rule R_i or machine M. If the two machines share monitored variables, the variable templates for the shared variables are intersected using the set intersection and the interval arithmetic described in Section 7.2.2. In theory, the intersection of the pre states could be empty, but this situation would occur only if the guards of the sub machine contradict the guards of the machine which uses the sub machine. If this situation is encountered, it would result in the rules of the sub machine never being enabled. Such a situation would not occur purposefully, otherwise there is no point in using the sub machine in the model.

The algorithm provided in Listing 7.4 is given for hierarchical composition for a single sub machine. It can easily be generalized for p sub machines SM_k used in the effect expression. For k machines that do not share variables, rules can be selected arbitrarily, one for each machine and can be aggregated into the test case template using the union operator. If the sub machines do not contain the same number of rules, some rule coverage will be repeated for some machines, but the combination of rules covered by each test case template will be unique. The total number of test case templates generated for k sub machines will be equal to the number of rules of the sub machine that has the largest number of rules. If the machines share monitored variables, the generated test case templates are assembled using the intersection operator for the pre state, following the approach of Listing 7.4, but for k machines. In this instance as well, the total number of generated test case templates is equal to the number of rules of the sub machine that has the largest number of rules.

Listing 7.4 Integration test case generation algorithm for a machine M and a sub machine SM. Machine M uses sub machine SM in the effect expression of rule R_i

- Generate the pre state $PreS_i$ for rule R_i of machine M using the approach given in Listing 7.2
  - If machine M and sub machine SM do not share variables:
    * Generate the post state $PostS_i$ for rule R_i of machine M, for the effect expressions not containing hierarchical composition, using the approach given in Listing 7.2
    * For each test case template $TCT_{SM,j}$ for sub machine SM:
    * Create m test case templates for machine M:
      · $TCT_i = \langle\ \langle PreS_i \cup PreS_{SM,j}, PostS_i \cup PostS_{SM,j}\rangle, \langle R_i, S_j\rangle\rangle$
  - If M and sub machine SM share monitored variables:
    * let $PreSS_{SM,j} \subseteq PreS_{SM,j}$ be the set of variable templates of machine SM for monitored variables which are shared with machine M
    * let $PreSN_{SM,j} = PreS_{SM,j} \setminus PreSS_{SM,j}$
    * let $PreSS_i \subseteq PreS_i$ be the set of variable templates of machine M for monitored variables which are shared with machine SM
    * let $PreSN_i = PreS_i \setminus PreSS_i$
    * calculate the new pre state $PreS_i' = (PreSS_i \cap PreSS_{SM,j}) \cup (PreSN_i \cup PreSN_{SM,j})$
    * Generate the post state $PostS_i$ for rule R_i of machine M, for the effect expressions not containing hierarchical composition, using the pre state $PreS_i'$ and the approach given in Listing 7.2
    * For each test case template $TCT_{SM,j}$ for sub machine: SM:
    * Create m test case templates for machine M:
      · $TCT_i = \langle\ \langle PreS_i', PostS_i \cup PostS_{SM,j}\rangle, \langle R_i, S_j\rangle\rangle$

7.4.2 Complexity Analysis

Generating the integration test case templates using the algorithm described in Listing 7.4 avoids the exponential growth that would result from generating a "flattened" version of the machine. Generating a flattened machine which uses multiple sub machines in its effect expression adds a number of rules to the host machine equal to the product of the number of rules of the sub machines. A coverage test suite template for the flattened machine would contain one test case per rule and hence an exponential number of test cases. For the algorithm given in Listing 7.4, the number of test cases necessary to cover the host machine and the sub machines is equal to the number of rules of the machine with the largest number of rules. The complexity for generating the pre state template of machine M is identical to the complexity of generating the pre state template for unit test cases, as explained in Section 7.4.2. Combining the test suite from machine M with the test suite from machine SM is linear in the number of rules for both machines. Calculating the post state template and the combination of the pre state template depends on the properties of the variables used in the rules.

7.4.3 Example

The example from Section 7.3.2 is extended to illustrate the generation of integration test cases. Two machines are added to the machine described in Listing 7.3. The first machine added, shown in Listing 7.5, does not contain hierarchical composition, and is analogous to the machine given in Listing 7.3. The `humidity` variable has a lower bound equal to 0 and an upper bound equal to 100. The test suite template for the machine of Listing 7.5 is given in Table 7.4 and has been obtained using the algorithm described in Listing 7.2.

The second machine which is added to the example uses both the machine from Listing 7.3 and the machine from Listing 7.5 in one of its rule effect expression. The machine is also adapted from the Timeliner case study with extensions to illustrate the generation of integration test cases. The machine of interest is shown in Listing 7.6. The coverage test case template for rule P_4 is easily generated using the algorithm

Listing 7.5 Rules of the SEQUENCE_HUMIDITY_MONITOR_WORK sub machine (partial)

```
S9: c4 -> c0 {
  t := 1950;

  if humid_seq_b = c4 and humidity > 39 then
    humid_seq_b := c0;
    humid_seq_s := done;
}

S10: c5 -> c5 {
  t := 1630;

  if humid_seq_b = c5 and humidity < 50 then
    humid_seq_b := c5;
    humid_seq_s := done;
}

S11: c5 -> c0 {
  t := 3195;

  if humid_seq_b = c5 and humidity >= 50 then
    humid_seq_b := c0;
    humidifier  := off;
    humid_seq_s := done;
}
```

Pre State	Post State	Coverage Item
$humid\_seq\_b\{c4\}$, $humidity = \{(39, 100]\}$	$humid\_seq\_b\{c0\}$ $humid\_seq\_s\{done\}$	S_9
$humid\_seq\_b\{c5\}$, $humidity = \{[0, 50]\}$	$humid\_seq\_b\{c5\}$, $humid\_seq\_s\{done\}$	S_{10}
$humid\_seq\_b\{c5\}$, $humidity = \{[50, 100]\}$	$humid\_seq\_b\{c0\}$, $humidifier\{off\}$, $humid\_seq\_s\{done\}$	S_{11}

Table 7.4: Template test suite for the machine of Listing 7.5

from Listing 7.2. The generated test suite is shown in Table 7.5 and contains 4 coverage test case templates. The test suite covers all the rules of the machines. The generated test suite shows the benefits of the integration testing strategy compared to generating a "flattened" machine and using the unit test case generation strategy. Generating an equivalent "flattened" machine would yield 10 rules, requiring 10 test cases to cover all the rules of the "flattened" machine. By using the integration test case generation algorithm described in Listing 7.4, 4 test cases are sufficient to cover all the rules of both machines.

Listing 7.6 Rules of the EXECUTE_PLANTSIM_SEQUENCES sub machine

```
P3: Execute sequences
{
  if exec_seq = not_done then
    SEQUENCE_HUMIDITY_MONITOR_WORK();
    SEQUENCE_TEMP_MONITOR_WORK();
    exec_seq    := done;
}

P4: Bundle finished
{
  if exec_seq = done then
    plantsim_s  := done;
    exec_seq    := not_done;
}
```

In order to add another level of hierarchical composition, which will prove useful in Section 7.6.4, an extra machine is introduced, which uses the machine shown in Listing 7.6 in rule V_1. The definition of the machine is shown in Listing 7.7. The test suite for the machine can easily be generated using the integration testing algorithm shown in Listing 7.4, combined with the test suite template shown in Table 7.5. The resulting test suite template is shown in Table 7.6.

7.5 Complete Test Case Generation Algorithm

The complete test case generation algorithm combines the unit test case generation approach from Section 7.3 and the integration test case generation strategy described in Section 7.4. The algorithm described in this section can be used on a machine at

Pre State	Post State	Coverage Item
$exec\_seq\{not\_done\}$, $humid\_seq\_b\{c4\}$, $humidity = \{(39, 100]\}$, $temp\_seq\_b\{b3\}$, $temperature = \{[-10, 19]\}$	$exec\_seq\{done\}$, $humid\_seq\_b\{c0\}$, $humid\_seq\_s\{done\}$ $temp\_seq\_b\{b4\}$, $heating\{on\}$, $temperature\{[6, 35]\}$	P_3, S_9, R_7
$exec\_seq\{not\_done\}$, $humid\_seq\_b\{c5\}$, $humidity = \{[0, 50]\}$, $temp\_seq\_b\{b4\}$, $temperature = \{[-10, 22)\}$	$exec\_seq\{done\}$, $humid\_seq\_b\{c5\}$, $humid\_seq\_s\{done\}$ $temp\_seq\_b\{b4\}$, $temp\_seq\_s\{done\}$	P_3, S_{10}, R_8
$exec\_seq\{not\_done\}$, $humid\_seq\_b\{c5\}$, $humidity = \{[50, 100]\}$, $temp\_seq\_b\{b4\}$, $temperature = \{[22, 40]\}$	$exec\_seq\{done\}$, $humid\_seq\_b\{c0\}$, $humidifier\{off\}$, $humid\_seq\_s\{done\}$ $temp\_seq\_b\{b0\}$, $heating\{off\}$, $temp\_seq\_s\{done\}$	P_3, S_{11}, R_9
$exec\_seq\{done\}$,	$exec\_seq\{not\_done\}$, $plantsim\_s\{done\}$,	P_4

Table 7.5: Template test suite for the machine of Listing 7.6

Listing 7.7 Rules of the PLANTSIM_BUNDLE sub machine

```
V1: Bundle Active
{
  if plantsim_bundle_status = active then
    EXECUTE_PLANTSIM_SEQUENCES();
}

V2: Bundle Inactive
{
  if plantsim_bundle_status = inactive then
    plantsim_s := done;
}
```

Pre State	Post State	Coverage Item
$exec\_seq\{not\_done\}$, $humid\_seq\_b\{c4\}$, $humidity = \{(39, 100]\}$, $temp\_seq\_b\{b3\}$, $temperature = \{[-10, 19]\}$, $plantsim\_bundle\_status\{active\}$	$exec\_seq\{done\}$, $humid\_seq\_b\{c0\}$, $humid\_seq\_s\{done\}$ $temp\_seq\_b\{b4\}$, $heating\{on\}$, $temperature\{[6, 35]\}$	V_1, P_3, S_9, R_7
$exec\_seq\{not\_done\}$, $humid\_seq\_b\{c5\}$, $humidity = \{[0, 50]\}$, $temp\_seq\_b\{b4\}$, $temperature = \{[-10, 22)\}$, $plantsim\_bundle\_status\{active\}$	$exec\_seq\{done\}$, $humid\_seq\_b\{c5\}$, $humid\_seq\_s\{done\}$ $temp\_seq\_b\{b4\}$, $temp\_seq\_s\{done\}$	V_1, P_3, S_{10}, R_8
$exec\_seq\{not\_done\}$, $humid\_seq\_b\{c5\}$, $humidity = \{[50, 100]\}$, $temp\_seq\_b\{b4\}$, $temperature = \{[22, 40]\}$, $plantsim\_bundle\_status\{active\}$	$exec\_seq\{done\}$, $humid\_seq\_b\{c0\}$, $humidifier\{off\}$, $humid\_seq\_s\{done\}$ $temp\_seq\_b\{b0\}$, $heating\{off\}$, $temp\_seq\_s\{done\}$	V_1, P_3, S_{11}, R_9
$exec\_seq\{done\}$, $plantsim\_bundle\_status\{active\}$	$exec\_seq\{not\_done\}$, $plantsim\_s\{done\}$	V_1, P_4
$plantsim\_bundle\_status\{inactive\}$	$plantsim\_s\{done\}$	V_2

Table 7.6: Template test suite for the machine of Listing 7.7

any level. The core idea behind the test case generation strategy is that a test suite can be associated with an individual machine to test the machine. The test cases are generated in a "bottom-up" fashion, by reusing the test suites already generated for machines which are combined through hierarchical composition. The complete test case generation algorithm is given in Listing 7.8.

Listing 7.8 Complete test case generation algorithm for a given TASM specification

- For a given TASM specification:
  - For each machine $\mathcal{M}_i$ which does not contain hierarchical composition:
    - Generate a coverage test suite template $TCTS_i$ using the algorithm described in Listing 7.2
    - Associate the test suite with the machine: $\langle \mathcal{M}_i, TCTS_i \rangle$
    - Add the pair to the test suite for the specification
  - For all remaining machines $\mathcal{N}_j$ which do not have a test suite associated with them:
    - Gather all machines $\mathcal{P}_k$ used for hierarchical composition in $\mathcal{N}_j$
    - Loop recursively until all machines $\mathcal{P}_k$ have an associated test suite in the model:
      - Generate a template coverage test suite, $TCST_k$, for machine $\mathcal{P}_k$ using the algorithm described in Listing 7.4
      - Associate the test suite with the machine: $\langle \mathcal{P}_k, TCTS_k \rangle$
      - Add the pair to the test suite for the specification
    - Generate a coverage test suite template, $TCST_j$ for machine $\mathcal{N}_j$ using the algorithm described in Listing 7.4
    - Associate the test suite with the machine: $\langle \mathcal{N}_j, TCTS_j \rangle$
    - Add the pair to the test suite for the specification
    - Loop until all machines are included in the test suite for the specification

7.5.1 Test Sequences

The test case generation strategy described in this chapter concerns the generation of test cases to exercise a single rule, and hence a single step of the specification. The assumption, as stated in Section 7.2, is that the specification can be executed in any state and that the state resulting from a step execution is fully observable. However, in practice, this is not necessarily the case since intermediate steps could be present between the consumption of inputs and the generation of outputs. For this reason, related approaches to test case generation often produce *test sequences*, which are sequences of inputs and outputs used to exercise the specification and the underlying system under test. The approach presented in this chapter could certainly be used to generate test sequences but provides a more flexible strategy to the execution of test cases. Since the test case templates are used to exercise a single rule, sequences could be generated by concatenating test case templates whose post states and pre states are congruent. Furthermore, a generic approach to test case generation could be devised by describing the properties of initial states and by describing the properties of observable states. Which such a theory, test sequences could be assembled using the concatenation of test case templates, with the pre state of the first test case template in the sequence meeting the property of the initial state and the final test case template in the sequence containing a post state which meets the observability criteria.

The test case generation strategy can be viewed as generating a set of ordered dominoes, where the top face is the pre state template and the bottom face is the post state templates. The dominoes can be assembled linearly to achieve a desired purpose, in the form of a test sequence. As long as all the test case templates from a given test suite are involved in a test sequence, the rule coverage criteria would still be preserved. These ideas are explored in the Timeliner case study in Section 8.5 and as part of future work in Chapter 9.

7.6 Regression Test Case Generation

Regression testing concerns the testing of an existing system which has already been tested to a certain extent, after changes have been made to either the system itself or to the specification. Conceptually, regression testing occurs after both unit and integration testing have been accomplished, to validate the correctness of a change made to the system or to the specification, during a late of the lifecycle [176]. Such a change could be the fixing of a defect, a change in requirements, or the introduction of new functionality. During regression testing, it is generally assumed that it is not feasible to complete repeat the unit testing and the integration testing efforts, due to time and budgetary constraints. If the unit and integration testing efforts can be repeated without too much toil, there is no need for a specific approach to the generation of regression test cases. The goal of regression testing is to identify a subset of both unit and integration testing that should be performed to adequately validate the change in the system or in the specification. In terms of a TASM specification and the concepts introduced so far, the regression test case generation strategy focuses on two aspects – which test cases need to be generated and/or modified to accommodate the change and which test cases need to be executed and/or repeated to validate the change. The goal of the strategy is to provide a minimal set of tasks that need to be performed to gain confidence into the correctness of the change, as opposed to repeating the entire testing activities for every single change that occurs. However, it is generally understood that the validation of the correctness of the changes could equally be achieved through repeating the unit and integration testing activities described in the previous sections, albeit at a higher cost.

The generation of regression test cases is achieved by combining the approaches for unit test case generation and for integration test case generation with the traceability approach explained in Chapter 6. The idea behind the test case generation strategy is to provide a mapping between the original specification and the modified specification, using the archetypical refinement types from Section 6.2.1. For each type of refinement, the correctness criteria explained in Section 6.2.2 are used to guide the generation of test cases. It is important to distinguish specification changes between defect correction and functionality addition. This distinction is important because defect correction purposefully alters the semantics and the goal of the change is not to preserve semantics. Consequently, the refinement would most likely be of the "any" variety. However, if the defect correction is limited to a single rule, the regression test case strategy can handle this case appropriately. For changes that span multiple facets of the model and where no refinement types can be applied, the generation of test cases would most likely need to be started anew using the algorithm of Section 7.5. In the following subsection, each type of refinement from Section 6.2.1 is listed and its effect on generation of regression test cases is analyzed.

7.6.1 Refinement Types

The regression test case generation strategy revolves around two basic concepts – adding/removing test cases to/from a test suite and modifying existing test cases. The specific approach depends on the type of refinement. Furthermore, the regression test case generation approach propagates changes upstream and downstream through hierarchical composition. When exploring the different types of refinement, 3 cases are considered. The first case considered is the situation where the change happens in a machine that is not involved in any sort of hierarchical composition. The second case to consider is the case where the modified machine is used for hierarchical composition, in the effect expression of another machine. Finally, the third case concerns the use of hierarchical composition in the rule of the machine where the change occurs. While these three cases are not mutually exclusive, they can be studied in isolation and the results can be generalized to a situation involving more than 1 of these situations. While the traceability approach presented in Chapter 6 aimed to provide an incremental approach to specification building, the regression test case strategy can be used as an incremental approach to test case generation. In the following subsections, it is assumed that unit test suites and integration test suites exist before the change is performed and the regression test cases are generated.

Step Expansion and Rule Expansion

As a reminder, the step expansion refinement is used to divide a step into multiple steps. Furthermore, the rule expansion refinement is used to modify an existing rule by adding items to the rule guard or to the effect expression. While both types of refinements capture different purposes, the regression test case generation strategy is the same for both of these types of refinements. As explained in Section 6.2.1, both types of refinements are one-to-many mappings where the rule of the original machine $\mathcal{M}_1$, R_i, is divided into m rules of a modified machine $\mathcal{M}'_1$, S_j $(1 \leqslant j \leqslant m)$. For the case where $\mathcal{M}_1$ is not involved in hierarchical composition, the coverage test case template that covers rule R_i is removed from the test suite, and m new coverage test case templates are generated using the approach described in Section 7.3, corresponding to the added rules in the modified machine. The rules S_j of machine $\mathcal{M}'_1$ are used as the coverage items. The new test case templates can be added to the test suite without any changes.

For the case where a modified rule S_j uses hierarchical composition in its effect expression, each test case template which covers rule R_i is removed from the test suite. Integration test case templates can be generated and added to the test suite using the approach described in Section 7.4 for each respective rule S_j.

For the case where the machine is used in the effect expression of another machine, $\mathcal{M}_2$, the test suite of machine $\mathcal{M}_2$ needs to be regenerated for the test case templates where rule R_i is in the coverage items. For each test case template containing rule R_i, m new coverage test case templates are generated, to cover the rule where the hierarchical composition occurs in machine $\mathcal{M}_2$ and the new rules S_j used for the step or rule expansion. The changes in the test suite of machine $\mathcal{M}_2$ are then propagated bottom-up, wherever a test case template covers a rule involved in the chain of changes. The algorithm which incorporates the three concepts is summarized in Listing 7.9.

Rule Addition

The test case generation strategy for the rule addition refinement is similar but slightly different than it is for the step and rule expansion refinements. Because the rule addition refinement introduces a new rule S_i which is unrelated to any rule in machine $\mathcal{M}_1$, the rule cannot already be involved in existing test case templates. Consequently, the rule addition refinement will introduce new test case templates, but does not need to remove or modify existing test case templates. If a rule is added to a machine, a test case template is generated for the added rule according to Listing 7.2 if the

Listing 7.9 Regression test case generation for a step or rule expansion refinement

- Generate test suite for machine M_1' by duplicating the test suite for machine M_1
- For rule R_i of machine M_1:
  - If rule R_i does not contain hierarchical composition:
    - Remove the test case template which covers rule R_i
    - Generate new test case templates for each refined rule S_j using the approach described in Listing 7.2
  - If rule R_i contains hierarchical composition:
    - For rules S_j which contain hierarchical composition, generate test case templates using the approach described in Listing 7.4
    - For rules S_j which do not contain hierarchical composition, generate test case templates using the approach described in Listing 7.2
    - Add all generated test case templates to the test suite for machine M_1'
  - If machine M_1 is used for hierarchical composition in another machine M_2:
    - Remove test case templates in machine M_1 where rule R_i is in the coverage item
    - For each rule D_k of machine M_2 which uses machine M_1 in its effect expression:
      - add m new test case templates to the test suite of machine M_2, using the approach described in Listing 7.4

added rule does not use hierarchical composition, and according to Listing 7.4 if the rule contains hierarchical composition. If machine $\mathcal{M}_1$ is used by another machine through hierarchical composition, coverage of rule S_j needs to be propagated upwards by adding a new test case template to each test suite associated with a machine which uses machine M_1 as a sub machine in one of its rule effect expressions. As is the case for the step and rule expansion refinements, when the modified machine is used for hierarchical composition the changes to the test cases need to be propagated upwards throughout the chain of hierarchical composition.

Step Contraction, Rule Contraction and Rule Deletion

For a step contraction refinement, a rule contraction refinement, and a rule deletion refinement, the regression test case generation strategy is different because rules are removed and hence test case templates must be removed. If the rules being removed do not contain hierarchical composition, the corresponding test case templates can be removed directly in the test suite of machine $\mathcal{M}_1$ and in all other test suites where a removed rule is part of the coverage criteria. However, if a removed rule contains hierarchical composition, the situation is more complex. For the rule deletion refinement, the corresponding test case templates can be removed directly because the hierarchical composition will no longer be part of machine $\mathcal{M}'_1$. The deletion of test case templates can be propagated upwards. A problem occurs if there are multiple levels of hierarchical composition involving machine $\mathcal{M}_1$, where the rule is removed. If multiple levels are present, removing the rule and the associated test case template upwards could cause the coverage test suite of a given machine to be incomplete. If this case occurs, the template test suite for the faulty machine can be generated anew, using the algorithm described in Listing 7.4. Furthermore, the test suites of all affected machines should also be regenerated fresh, using the same algorithm.

The Any Refinement

Since the "any" refinement is a type of refinement used for refinements that do not fit the listed types, the mapping is defined as a many-to-many mapping. Because there is no clear structure for the "any" refinement, regression testing strategies cannot be devised. If the arity of the refinement is analogous to the arity of the refinements mentioned previously, the strategies previously expressed could be used. However, if the arity of the refinement does not match the cases mentioned previously, the test case generation must start anew, using the algorithm described in Listing 7.8, invalidating all machine test suites affected by the "any" refinement.

7.6.2 Test Case Execution

The primary purpose of the regression testing strategy is to modify existing test suites to accommodate the changes introduced in the specification. In what has been discussed so far, the goal of the approach is to minimize the amount of test case generation that needs to be performed, if unit and integration test suites already exist. However, another important facet of regression testing is identifying which test cases need to be executed to fully exercise the changes introduced in the specification. Given the structure of the test case generation strategy expressed in Section 7.3 and in Section 7.4, the test suite associated with each machine contains coverage criteria for each test case template by listing covered rules. Determining which test cases need to be executed to exercise the change is fairly straightforward because the influenced rules can be easily identified through the coverage criteria. Consequently, by using the traceability approach from Chapter 6, and the regression test case generation strategy described in the previous section, the test cases that need to be executed can be easily identified.

7.6.3 Complexity Analysis

The worst-case scenario for the generation of regression test cases occurs when all the test suites of all the machines need to be regenerated, leading to the complexity of the complete test case generation algorithm described in Listing 7.8. The complexity of that algorithm builds on the complexity of the unit test case generation algorithm and on the complexity of the integration test case generation.

For test case generation where only a subset of the test suite needs to be regenerated, the complexity of the test case generation will depend on the number of affected rules and of affected test case templates. In turn the number of affected test case templates will depend on the hierarchical composition properties of the TASM specification. In the worst case for hierarchical composition, the affected number of test cases will vary linearly with the number of machines in the specification and the number of modified/added rules in the changes made to the specification.

7.6.4 Example

The example from Section 7.4.3 is modified to demonstrate the test case generation strategy for regression testing. The machine shown in Listing 7.6 is refined through a step expansion refinement into two rules. The resulting machine is shown in Listing 7.10. In Listing 7.10, rule P_3 from Listing 7.6 is refined into two rules, rule D_3 and rule D_4. The traceability relationship for the refinement can be expressed as:

$$\begin{aligned} \mathrm{T} = \mathbb{T}_{sexp} \cup \mathbb{T}_{id} &= (\langle\{P_3\},\ \{D_3,\ D_4\}\rangle) \cup (\langle\{P_4\},\ \{P_4\}\rangle) \\ &= (\langle\{P_3\},\ \{D_3,\ D_4\}\rangle,\ \langle\{P_4\},\ \{P_4\}\rangle) \end{aligned}$$

For all other rules of all other machines in the example, the traceability is achieved through identity refinements.

Given the approach described in Section 7.6.1 for the step expansion refinement, only the test cases involving the P_3 rule as a coverage item need to be regenerated and executed. The test suite of machine *SEQUENCE_HUMIDITY_MONITOR_WORK*, shown in Table 7.3, and the test suite of the machine *SEQUENCE_TEMP_MONITOR_WORK*, shown in Table 7.4 do not need to be modified. For the *EXECUTE_PLANTSIM_SEQUENCES*, only the last row of the test suite shown in Table 7.5,

Listing 7.10 Rules of the refined EXECUTE_PLANTSIM_SEQUENCES sub machine

```
D3: Execute sequences
{
  if exec_seq = not_done and seq = humid then
    SEQUENCE_HUMIDITY_MONITOR_WORK();
    seq        := temp;
}

D4: Execute sequences
{
  if exec_seq = not_done and seq = temp then
    SEQUENCE_TEMP_MONITOR_WORK();
    exec_seq   := done;
    seq        := humid;
}

P4: Bundle finished
{
  if exec_seq = done then
    plantsim_s := done;
    exec_seq   := not_done;
}
```

Pre State	Post State	Coverage Item
$exec\_seq\{not\_done\}$, $seq\{humid\}$, $humid\_seq\_b\{c4\}$, $humidity = \{(39, 100]\}$	$humid\_seq\_b\{c0\}$, $humid\_seq\_s\{done\}$, $seq\{temp\}$	D_3, S_9
$exec\_seq\{not\_done\}$, $seq\{humid\}$, $humid\_seq\_b\{c5\}$, $humidity = \{[0, 50]\}$	$humid\_seq\_b\{c5\}$, $humid\_seq\_s\{done\}$, $seq\{temp\}$	D_3, S_{10}
$exec\_seq\{not\_done\}$, $seq\{humid\}$, $humid\_seq\_b\{c5\}$, $humidity = \{[50, 100]\}$	$humid\_seq\_b\{c0\}$, $humid\_seq\_s\{done\}$, $seq\{temp\}$, $humidifier\{off\}$	D_3, S_{11}
$exec\_seq\{not\_done\}$, $seq\{temp\}$, $temp\_seq\_b\{b3\}$, $temperature = \{[-10, 19]\}$	$exec\_seq\{done\}$, $temp\_seq\_b\{b4\}$, $heating\{on\}$, $temperature\{[6, 35]\}$, $seq\{humid\}$	D_4, R_7
$exec\_seq\{not\_done\}$, $seq\{temp\}$, $temp\_seq\_b\{b4\}$, $temperature = \{[-10, 22)\}$	$exec\_seq\{done\}$, $temp\_seq\_b\{b4\}$, $seq\{humid\}$	D_4, R_8
$exec\_seq\{not\_done\}$, $seq\{temp\}$, $temp\_seq\_b\{b4\}$, $temperature = \{[22, 40]\}$	$exec\_seq\{done\}$, $temp\_seq\_b\{b0\}$, $heating\{off\}$, $seq\{humid\}$	D_4, R_9
$exec\_seq\{done\}$	$exec\_seq\{not\_done\}$, $plant\_sim\_s\{done\}$,	P_4

Table 7.7: Test suite template for the machine of Listing 7.10

the test case template which covers rule P_4, does not need to be regenerated. The test case templates to cover the refined rules, rule D_3 and rule D_4, are regenerated using the algorithm given in Listing 7.4. The resulting test suite is given in Table 7.7. It is important to note that the *EXECUTE_PLANTSIM_SEQUENCES* machine is used as a sub machine in a rule effect expression of machine *PLANTSIM_BUNDLE*, in rule V_1, as shown in Listing 7.7. Consequently, the changes in the test suite for the *EXECUTE_PLANTSIM_SEQUENCES* need to be propagated in the test suite of the *PLANTSIM_BUNDLE* machine. The resulting test suite for the *PLANTSIM_BUNDLE* machine is shown in Table 7.8.

Pre State	Post State	Coverage Item
$exec\_seq\{not\_done\}$, $seq\{humid\}$, $humid\_seq\_b\{c4\}$, $humidity = \{(39, 100]\}$, $plantsim\_bundle\_status\{active\}$	$humid\_seq\_b\{c0\}$, $humid\_seq\_s\{done\}$, $seq\{temp\}$	V_1, D_3, S_9
$exec\_seq\{not\_done\}$, $seq\{humid\}$, $humid\_seq\_b\{c5\}$, $humidity = \{[0, 50]\}$, $plantsim\_bundle\_status\{active\}$	$humid\_seq\_b\{c5\}$, $humid\_seq\_s\{done\}$, $seq\{temp\}$	V_1, D_3, S_{10}
$exec\_seq\{not\_done\}$, $seq\{humid\}$, $humid\_seq\_b\{c5\}$, $humidity = \{[50, 100]\}$, $plantsim\_bundle\_status\{active\}$	$humid\_seq\_b\{c0\}$, $humid\_seq\_s\{done\}$, $seq\{temp\}$, $humidifier\{off\}$	V_1, D_3, S_{11}
$exec\_seq\{not\_done\}$, $seq\{temp\}$, $temp\_seq\_b\{b3\}$, $temperature = \{[-10, 19]\}$ $plantsim\_bundle\_status\{active\}$	$exec\_seq\{done\}$, $temp\_seq\_b\{b4\}$, $heating\{on\}$, $temperature\{[6, 35]\}$, $seq\{humid\}$	V_1, D_4, R_7
$exec\_seq\{not\_done\}$, $seq\{temp\}$, $temp\_seq\_b\{b4\}$, $temperature = \{[-10, 22)\}$, $plantsim\_bundle\_status\{active\}$	$exec\_seq\{done\}$, $temp\_seq\_b\{b4\}$, $seq\{humid\}$	V_1, D_4, R_8
$exec\_seq\{not\_done\}$, $seq\{temp\}$, $temp\_seq\_b\{b4\}$, $temperature = \{[22, 40]\}$, $plantsim\_bundle\_status\{active\}$	$exec\_seq\{done\}$, $temp\_seq\_b\{b0\}$, $heating\{off\}$, $seq\{humid\}$	V_1, D_4, R_9
$exec\_seq\{done\}$, $plantsim\_bundle\_status\{active\}$	$exec\_seq\{not\_done\}$, $plantsim\_s\{done\}$	V_1, P_4
$plantsim\_bundle\_status\{inactive\}$	$plantsim\_s\{done\}$	V_2

Table 7.8: Test suite template for the machine of Listing 7.7

Machine	Test Case
EXECUTE_PLANTSIM_SEQUENCES	D_3, S_9 D_4, R_7
PLANTSIM_BUNDLE	V_1, D_3, S_{10} V_1, D_4, R_9

Table 7.9: List of test cases that need to be executed to cover the refinement

Test Case Execution

Based on the generation of regression test cases, only a subset of the test case templates need to be executed to validate the correctness of the change. The set of test cases which need to be executed are shown in Table 7.9, for each machine affected by the change. Because there was no change to the S_i rules and to the R_i rules, only rule D_3 and rule D_4 need to be exercised in machine *EXECUTE_PLANTSIM_SEQUENCES*. A test case template to cover each rule was selected arbitrarily from Table 7.7. A similar approach is used for machine *PLANTSIM_BUNDLE*, where only 2 test case templates need to be executed to validate the effect of the change.

7.7 Segue into Chapter 8

This chapter presented an approach to automatically generate test cases based on a specification expressed in the TASM language. More specifically, facilities were presented to automatically generate unit test cases, integration test cases, and regression test cases. The test case generation capabilities represent the final feature of the proposed framework. In the next chapter, Chapter 8, experimentation using the presented framework is performed using three case studies.

Chapter 8

Case Studies

This chapter presents the results of the three case studies that are used to evaluate the capabilities of the presented framework. The applications used for the case studies are explained in details in Section 2.8. In this chapter, each case study is presented in a separate section and each section presents the TASM specification of the case study, the functional analysis results, the execution time analysis results, the resource consumption analysis results, and the test case generation results. In addition, he Electronic Throttle Controller (ETC) case study, analyzed in Section 8.2, in Section 8.3, and in Section 8.4, utilizes the bi-directional traceability strategy by relating three separate models of the ETC – a high level model, and tasking model, and a low level model. The ETC case study is also used to demonstrate the test case generation approach for regression testing. The other case studies are used to demonstrate the unit and integration test case generation capabilities of the framework.

For each case study, the TASM model is described in each respective section, but only a subset of the model listings are provided. The complete TASM models for each case study are provided in the appendices. The model for the production cell case study is provided in Appendix D, the model for the ETC case study is provided in Appendix E, and the model for the Timeliner case study is provided in Appendix F. Furthermore, for each case study, the UPPAAL model obtained through the translations is described, but the resulting complete UPPAAL model is not included, for brevity. Each case study is followed by a brief discussion of the results and a commentary on the practical usefulness of the framework features. The overall evaluation of the framework, in light of the results of the case studies, is presented in Chapter 9. Chapter 9 also recapitulates the contributions of the thesis in light of the research objectives presented in Chapter 1.

8.1 Production Cell

The production cell case study is an automated manufacturing system which is based on an industrial plant in Karlsruhe in Germany [163]. The case study is described in details in Section 2.8.1. As a reminder, the logical view of the production cell is provided in Figure 8-1 and contains 5 hardware components to achieve the system's goals – a loader, a feed belt, a robot, a press, and a deposit belt. The embedded controller must command each component to stamp blocks, which are introduced in the system by the loader. The controller reads the state of the system through a set of sensors, listed in Table 2.4 and commands the various hardware components through a set of actuators, listed in Table 2.2.

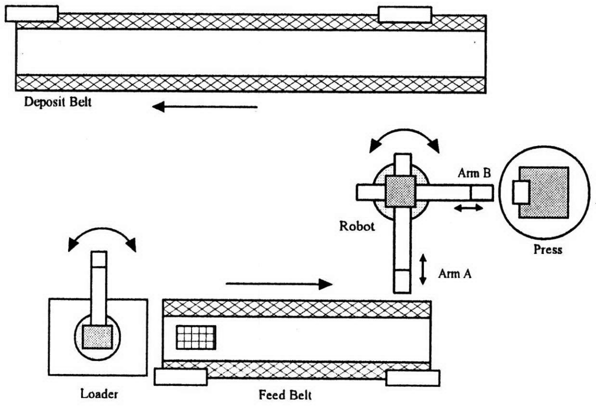


Figure 8-1: Top view of the production cell

8.1.1 Model

The TASM model of the production cell is described in great detail for this case study, because it is the first case study presented. The models pertaining to the other

Name	Type	Purpose
Controller	Main	Commands the actuators
Loader	Main	Loads blocks onto the feed belt
Feed	Main	Carries blocks from the loader to the robot
Robot	Main	Simulates the rotation of the robot
ArmA	Main	Simulates arm a
ArmB	Main	Simulates arm b
Press	Main	Stamps blocks
Deposit	Main	Carries blocks out of the system

Table 8.1: List of main machines used in the production cell model

case studies are described in less details. In the production cell TASM model, each component of the production cell is modeled as a main machine, except for the robot. As a reminder, in the TASM language, a main machine is a unit of concurrency. The robot component is modeled as three separate main machines to capture the parallel behavior of the motion base, arm a, and arm b, all of which can be commanded independently. Sub machines and function machines are used, mostly to structure the actions of the controller. The complete list of main machines is shown in Table 8.1. In the following sub sections, as each main machine is explained, the sub machines and function machines that are used in the model are given. The complete list of all machines used in the production cell case study model, is available in Table D.1 in Appendix D.

The Environment

As a reminded, in a TASM model, the environment contains the list of user-defined types, the list of global variables, and the list of resources used in the model. The list of user-defined types used in the production cell model is given in Listing 8.1. The `status` type is used to keep track of whether various parts of the system (e.g., the belts, the arms, and the press) are loaded or empty. The `armposition` type is used to represent the position of the arms with respect to the robot angle, in discrete steps. For example, if arm a is at the feed or at the press, the controller takes certain actions. If arm a is neither at the feed nor at the press, the arm is in transit. This "discretization" is used because if an arm is not at the press, at the deposit belt, or at the feed belt, it makes no difference to the controller whether the robot angle is 31, 32, or 33 degrees. The discrete positions of the arms were obtained through the specification of the desired behavior of the controller in the problem definition [163].

The models for the arms use a slightly different approach than the rotation of the robot base. Instead of relating a continuous length to a set of discrete values, two discrete values of interest are used via the `armextension` type - `retracted` and `extended`. The `Actuator` type is used to indicate whether a motor or a magnet is on or off. The `Polarity` type is used to set the polarity of the various motors. The `Stamp` data type is used to set the block status in the press. Finally, the `Error` type is used to catch certain types of errors when performing safety analysis of the controller. The use of the `Error` type and the topic of functional analysis and verification are treated in Section 8.1.2.

Listing 8.1 User-defined types of the production cell model

```
status          := {empty, loaded};
armposition     := {atfeed, atpress, atdeposit, intransit};
armextension    := {retracted, extended};
Actuator        := {on, off};
Polarity        := {positive, negative};
Stamp           := {notfinished, finished};
Error           := {none, invaliddrop, invalidpickup};
```

The user-defined types of the model are used to restrict the set of values that the variables of the system can take. A subset of the variables that are used in the model, with their associated initial conditions, are shown in Listing 8.2. The complete list of variables is given in Listing D.2 and in Listing D.3 in Appendix D. The variables are grouped into *sensors*, which correspond to the sensors of Table 2.4, *actuators*, which correspond to the actuators of Table 2.2 and Table 2.3, *constants*, and *redundant information*. The *redundant information* is used to keep track of the system's state, inside the software, as the controller performs actions. For example, the feed belt is loaded once the loader puts a block on it and stays loaded until the robot picks up the block. The `loaded_blocks` and the `processed_blocks` variables are used to keep track of how many blocks have been inserted in the system and how many blocks have exited the system. The `wait` variable and the `robot_wait` variable are used to synchronize the controller and the robot rotation. Essentially, they are used to enforce *fairness* [30], to make sure that the system's state can progress and that the robot can process a command from the controller. More specifically, since the controller actions are instantaneous, the `wait` variable is used to enable the environment to make progress between controller actions. Without this constraint, the controller could perform an infinite number of actions before other components get a chance to perform a single action, leading to so-called *Zeno runs* [30].

The convention of the `robot_angle` variable is that it is 0 when arm a is at the feed and arm b is at the press, as in Figure 8-1. As the robot rotates counter clockwise, the angle increases by 30 degrees. When the value of the `robot_angle` variable is 90, arm a is at the press and arm b is at the deposit. For the controller strategy used in this model, the value of the `robot_angle` variable will remain between 0 degrees and 90 degrees inclusively. The model also contains one resource, *power*, which gets consumed when the hardware components are operating.

Throughout the model, as a convention, capital letters are used to describe constants and sub machine calls. The `ROTATION_ANGLE` constant is used as the discrete angle by which the robot is rotated when the `motor_robot` actuator is `on`. The value "30" was selected as the delta of rotation because it fits the problem description of the durative actions listed in Table 2.5.

In the following subsections, the main components, modeled as main machines, are described one by one. The description of the *ArmB* main machine and of the *Deposit* main machine are omitted because they are similar to the *ArmA* main machine and to the *Feed* main machine, respectively. The complete TASM model for the production cell case study is available in Appendix D.

Loader

The loader is the component that drives the system by putting blocks on the feed belt. The rules of the *Loader* main machine are shown in Listing 8.3. In the listing, the variable `number` is an integer variable that is internal to the *Loader* machine. This variable is used in the constructor to determine how many blocks the loader will

Listing 8.2 Variables of the production cell model (partial)

```
//sensors
Integer[0, 90]  robot_angle        := 0;
Stamp           press_block        := notfinished;
Boolean         feed_begin         := False;
armextension    armaext            := retracted;
armextension    armbext            := retracted;
Boolean         feed_end           := False;
Boolean         deposit_begin      := False;
Boolean         deposit_end        := False;

//redundant info, derivable from sensors
armposition     armapos            := atfeed;
armposition     armbpos            := atpress;
status          arma               := empty;
status          armb               := empty;
status          feed_belt          := empty;
status          deposit_belt       := empty;
status          press              := empty;

//other variables
Boolean         wait               := False;
Boolean         robot_wait         := False;
Integer[0, 50]  loaded_blocks      := 0;
Integer[0, 50]  processed_blocks   := 0;
Boolean         loader_done        := False;
Error           error              := none;
armposition     robot_destination  := atfeed;

//actuators
Actuator        motor_press        := off;
Actuator        motor_arma         := off;
Actuator        motor_armb         := off;
Actuator        magnet_arma        := off;
Actuator        magnet_armb        := off;
Actuator        motor_robot        := off;
Actuator        motor_feed         := off;
Actuator        motor_deposit      := off;

Polarity        motor_press_p      := positive;
Polarity        motor_arma_p       := positive;
Polarity        motor_armb_p       := positive;
Polarity        motor_robot_p      := positive;
Polarity        motor_feed_p       := positive;
Polarity        motor_deposit_p    := negative;
```

insert in the system. The first rule, rule *R1*, loads blocks on the feed belt as soon as the feed belt is empty. Per the properties of the actions listed in Table 2.5, loading a block on the belt takes 2 time units and consumes 200 units of power. Once the action is complete, the feed belt is loaded and the *feed_begin* sensor is set to true, to notify the controller that there is a block on the feed belt.

Listing 8.3 Rules of the *Loader* main machine

```
R1: The feed belt is empty, put a block on it
{
  t     := 2;
  power := 200;

  if loaded_blocks < number - 1 and feed_belt = empty then
    feed_belt       := loaded;
    loaded_blocks   := loaded_blocks + 1;
    feed_begin      := True;
}

R2: This is the last block...
{
  t     := 2;
  power := 200;

  if loaded_blocks = number - 1 and feed_belt = empty then
    feed_belt       := loaded;
    loaded_blocks   := loaded_blocks + 1;
    feed_begin      := True;
    loader_done     := True;
}

R3: The feed belt is loaded, do nothing
{
  t     := next;

  if feed_belt = loaded and loaded_blocks < number then
    skip;
}
```

Rule *R2* is used to put the last block on the feed belt and to notify the controller that the loader will no longer put blocks on the feed belt, through the `loader_done` variable. The last rule, rule *R3*, is used to wait and elapse time until the next state change. The "`t := next`" construct is used to keep the machine alive until a change to monitored variables occurs. Once all blocks have been loaded in the system, no rule will be enabled for the *Loader* machine and the machine will stop, per the semantics of the TASM language described in Section 4.3.

Feed Belt

The *Feed* main machine is a simple machine that contains only two rules. The rules of the machine are shown in Listing 8.4. Rule *R1* is the only rule that changes the state. The rule is enabled when there is a block on the belt, that block is at the beginning of the belt, the motor is on, and the polarity of the motor is positive. When this condition is met, the rule will take 5 time units to complete and will consume 500 units of power, per the description in Table 2.5. The effect of executing this rule will be such that the block will move from the beginning of the feed belt to the end, and the appropriate state change is reflected in the sensors *feed_begin* and *feed_end* by setting the appropriate variables.

Listing 8.4 Rules of the *Feed* main machine

```
R1: Block goes to end of belt
{
  t     := 5;
  power := 500;

  if feed_belt = loaded and feed_begin = True and
     motor_feed = on and motor_feed_p = positive then
    feed_begin  := False;
    feed_end    := True;
}

R2: Else
{
  t := next;

  else then
    skip;
}
```

Rule *R2* will be enabled and fired whenever rule *R1* is not enabled. Rule *R2* has no effect on the environment and is used solely to keep the machine running. Once again, the "`t := next`" construct is used to indicate that the machine will not perform any steps until a change to its monitored variables occurs.

Press

The *Press* main machine is similar to the *Feed* main machine. It simply reacts to the motor being on and causes the state change to take place once the stamping of the block is completed. The rules of the machine are shown in Listing 8.5.

Listing 8.5 Rules of the *Press* main machine

```
R1: Press is loaded, motor is on
{
  t     := 11;
  power := 1500;

  if motor_press = on and press = loaded and press_block = notfinished then
    press_block := finished;
}

R2: Else
{
  t := next;

  else then
    skip;
}
```

The *Deposit* main machine is also similar to the *Feed* main machine shown in Listing 8.4. It is interesting to note that the three components described so far update the state only through the sensors and react to state changes only through the information available through actuator values. The *Loader* main machine is a bit different than the other machines explained so far because it is used to drive the system and is an active component, in contrast to the feed, press, and deposit which are purely reactive components.

Robot

The *Robot* main machine is used to describe the rotation of the base of the robot. The machine, whose rules are shown in Listing 8.6, uses the `robot_wait` variable to give a chance for the controller to stop the motor before rotation resumes. This behavior could also have been enforced by the use of a communication channel. The *Robot* main machine differs from other machines described so far because it uses a sub machine called *ROBOT_MOTION*. As a refresher, a sub machine is a unit of hierarchical composition. The behavior of the main machine is defined in terms of the sub machine by merging the update set yielded by the sub machine with update sets yielded by other sub machines, if applicable and with other updates to variables included in the rule effect expression. For rule *R1*, the updates to variables yielded by the *ROBOT_MOTION* sub machine will be merged with the update to the `robot_wait` variable. Since rule *R1* does not have a time or resource annotation, the duration and resource consumption of the rule execution are defined by the sub machine annotations, if they are present. In the case of the *ROBOT_MOTION* sub machine, the machine contains time and resource annotations, which will be used to determine the time and resource behavior of the *Robot* main machine.

Listing 8.6 Rules of the *Robot* main machine

```
R1: do
{
  if robot_wait = False then
    ROBOT_MOTION();
    robot_wait := True;
}

R2: wait
{
  t := next;

  if robot_wait = True then
    robot_wait := False;
}
```

The use of a sub machine can be viewed as a nested *if* statement. Sub machines are nothing more than syntactic sugar to help structure specifications, as explained in Theorem 4.2. The rules of sub machine *ROBOT_MOTION* are shown in Listing 8.7.

In Listing 8.7, rules *R1* and *R2* are used to rotate the robot clockwise and counter clockwise depending on the polarity of the motor. Rule *R1* of the sub machine uses two function machines, *rotateClockwise* and *armPosition*. In the TASM language, function machines are macros that are analogous to functions in programming languages. Function machines have no side-effect in that they do not change environment variables. The *rotateClockwise* function is used to return the resulting angle of doing

Listing 8.7 Rules of the *ROBOT_MOTION* sub machine

```
R1: rotate clockwise
{
  t          := 2;
  power      := 1000;

  if motor_robot = on and motor_robot_p = negative then
    robot_angle := rotateClockwise();
    armapos     := armPosition(ARM_A_FEED_ANGLE, ARM_A_DEPOSIT_ANGLE,
                               ARM_A_PRESS_ANGLE, rotateClockwise());
    armbpos     := armPosition(ARM_B_FEED_ANGLE, ARM_B_DEPOSIT_ANGLE,
                               ARM_B_PRESS_ANGLE, rotateClockwise());

}

R2: rotate counterclockwise
{
  t          := 2;
  power      := 1000;

  if motor_robot = on and motor_robot_p = positive then
    robot_angle := rotateCounterClockwise();
    armapos     := armPosition(ARM_A_FEED_ANGLE, ARM_A_DEPOSIT_ANGLE,
                               ARM_A_PRESS_ANGLE, rotateCounterClockwise());
    armbpos     := armPosition(ARM_B_FEED_ANGLE, ARM_B_DEPOSIT_ANGLE,
                               ARM_B_PRESS_ANGLE, rotateCounterClockwise());
}

R3: Else
{
    else then
      skip;
}
```

one rotation step. The function machine also ensures that the angle doesn't go below 0 or over 360. Essentially, it returns *(robot_angle + ROTATION_ANGLE) modulo 360.* The *armPosition* function machine is used to set the position of each arm based on the resulting robot angle. Since the *robot_angle* will not be updated until after the rule has been completed, the *armPosition* function machine needs to anticipate what the robot angle will be, which explains the call to the *rotateClockwise* function machine as a parameter. The rules of the function machine *armPosition* are shown in Listing 8.8. The durations and power consumptions used for the robot rotation, in rules *R1* and *R2*, are in accordance with the problem definition given in Table 2.5. For the *armPosition* function machine, the new robot angle is passed in through the *value* parameter. The other parameters include *feed_angle*, *deposit_angle*, and *press_angle*. These values are used to determine whether the rotation will result in a given arm being at the feed, at the press, at the deposit, or in transit.

Listing 8.8 Rules of the *armPosition* function machine

```
R1: CCW rotation will put arm at feed
{
  if value = feed_angle then
    out := atfeed;
}

R2: CCW rotation will put arm at deposit
{
  if value = deposit_angle then
    out := atdeposit;
}

R3: CCW rotation will put arm at press
{
  if value = press_angle then
    out := atpress;
}

R4: Else, CCW rotation will put arm in transit
{
  else then
    out := intransit;
}
```

Arm A

The *ArmA* main machine is used to simulate the behavior of arm a. The action that arm a can perform include extending, retracting, picking up a block, dropping a block. The main machine uses two sub machines, *DROP_ARM_A* and *PICK_UP_ARM_A*.

Listing 8.9 Rules of the *ArmA* main machine

```
R1: Extend arm
{
  t     := 3;
  power := 1200;

  if motor_arma = on and motor_arma_p = positive and
     armaext = retracted then
    armaext := extended;
}

R2: Retract arm
{
  t     := 2;
  power := 1100;

  if motor_arma = on and motor_arma_p = negative and
     armaext = extended then
    armaext := retracted;
}

R3: Pick up block
{
  if magnet_arma = on and arma = empty and
      armapos = atfeed and feed_end = True then
    PICK_UP_ARM_A();
}

R4: Drop block
{
  if magnet_arma = off and arma = loaded then
    DROP_ARM_A();
}

R5: Else
{
  t := next;

  else then
    skip;
}
```

The rules of the *DROP_ARM_A* sub machine are shown in Listing 8.10. The sub machine is interesting because it uses the `error` variable to communicate an erroneous state. More specifically, the sub machine will set the `error` variable to `invaliddrop` if the controller commands the arm to drop a block and the arm is not extended, if the controller commands the arm to load the press and the press is already loaded, or if the controller commands the arm to drop a block while the arm is in transit. The controller should not command the magnet to drop a block under these conditions. Using the variable is not necessary to detect that an erroneous state is reachable, but it illustrates the clever use of rules. The safety requirements to ensure that blocks are not dropped under undesirable conditions could be phrased using the value of the variable, such as "the value of the `error` variable is never equal to `invaliddrop`".

Listing 8.10 Rules of the DROP_ARM_A sub machine

```
R1: Drop at press
{
  t     := 2;
  power := 800;

  if armapos = atpress and arma = loaded and
     armaext = extended and press = empty then
    arma            := empty;
    press           := loaded;
    press_block     := notfinished;
}

R2: Invalid drop
{
  if armapos != atpress or arma = empty or
     press = loaded or armaext != extended then
    error := invaliddrop;
}
```

As a reminder, rule *R4* of Listing 8.9 does not contain time or resource consumption annotations. Consequently, the duration and resource consumption of the rule execution will come from the *DROP_ARM_A* sub machine, since that machine contains time and resource annotations.

Controller

The *Controller* main machine is the most complex machine of the model. In a fashion similar to the *Robot* main machine, the *Controller* machine uses a variable called wait to enable the environment to make progress before performing an action. For the controller, this waiting is necessary because all the actions of the controller are instantaneous and the environment must be given a chance to make progress. Otherwise, the controller could perform an infinite number of steps before an environment change happens. In real-time system terms, the *Controller* main machine can be viewed as a sporadic task which gets released whenever a sensor value changes. The rules of the Controller main machine, shown in Listing 8.11, make heavy use of sub machines. The semantics of sub machines and hierarchical composition are such that all sub machines operate in parallel and the resulting update sets of each machine are composed with one another. The commanding of all of the actuators are performed independently, in parallel.

Listing 8.11 Rules of the *Controller* main machine

```
R1: Issue Commands
{
  if wait = False then
    OPERATE_FEED();
    OPERATE_DEPOSIT();
    OPERATE_ROBOT();
    OPERATE_ARM_A();
    OPERATE_ARM_B();
    OPERATE_PRESS();
    wait := True;
}

R2: Wait for a step
{
  t := next;

  else then
    wait := False;
}
```

The rules of the *OPERATE_DEPOSIT* sub machine are shown in Listing 8.12. The listing shows how the controller uses only sensor values to interpret the state of the system, and uses only the actuators to command the various components of the system. Listings for other sub machines of the "*OPERATE_ABC*" nature are similar and are given in Appendix D.

Listing 8.12 Rules of the *OPERATE_DEPOSIT* sub machine

```
R1: turn on motor
{
  if motor_deposit = off and deposit_begin = True then
    motor_deposit_p     := negative;
    motor_deposit       := on;
}

R2: turn off motor
{
  if motor_deposit = on and deposit_end = True then
    motor_deposit := off;
}

R3: nothing to do
{
  else then
    skip;
}
```

Complete Model

The complete production cell TASM model contains 8 main machines, one for each component shown in Figure 8-1, and one for the controller. The model also contains 3 function machines, and 16 sub machines. The complete production cell model is documented in Appendix D where the list of all machines is given in Table D.1.

8.1.2 Functional Analysis

The purpose of the production cell case study, as outlined in [163], is to evaluate and compare different formal methods. Part of the problem definition is to understand how different approaches model and prove properties of the production cell case study. Some of the properties that should be proved include restrictions on the commands that the controller sends out to the hardware components. For example, the controller shall not command the robot to drop blocks in places other than the press and the deposit belt. Furthermore, the robot should never be rotated when the arms are extended. In order to verify these properties in the TASM model, model checking presents a natural fit since the model is finite and the safety properties can be easily formulated as temporal logic properties over the variable values of the model.

Safety and Liveness Properties

For this case study, three safety properties are verified using a model checking approach. The safety properties are verified using two different strategies. The first strategy uses a simple *safety invariant* property over the variable values of the model, to express that "a certain state shall never be reached". Two safety properties are verified using this strategy. The first safety property states that "the robot shall not rotate while an arm is extended". The second property states that "arm a shall only be extended at the press and at the feed belt" The temporal logic formulas corresponding to the two properties are shown below:

- $A\ G\ (motor\_robot = on) \rightarrow (arma = retracted \wedge armb = retracted)$

- $A\ G\ (arma = extended) \rightarrow (armapos = atpress \vee armapos = atfeed)$

The first temporal logic formula states that it is always true in the model that whenever the robot motor is on, the arms are retracted. The second temporal logic formula states that it is always true in the model that whenever arm a is extended, arm a is at the press or arm a is at the feed. The second strategy to verify safety properties involves embedded error values inside of the model, in a manner analogous to assertions in programming languages [132]. In the TASM model, the user-defined type `Error` in Listing 8.1 is used to create errors that can be embedded in the model. For example, in Listing 8.10, the `error` variable is set to `invaliddrop` if the controller tries to drop a block from arm a at a place other than the press. Similar rules were added to the *DROP_ARM_B*, *PICK_UP_ARM_A*, and *PICK_UP_ARM_B* sub machines. The first safety property wraps a set of safety assertions, including "the robot shall not drop blocks in places other than the press and the deposit belt", "the robot shall not drop a block in the press if the press is already loaded", and "the robot shall not drop a block if the arm is not extended". The second safety property also wraps a set of safety assertions similar to the first safety property, but concerns the picking up of blocks. The included properties in the assertion include "the robot shall not pick up blocks in places other than the press and the feed belt". These two properties can be easily stated using the `error` variable and expressing the assertion as a simple safety invariant on the values of the variable:

- *A G error != invaliddrop*

- *A G error != invalidpickup*

The temporal logic formulas state that the `error` variable is never set to `invaliddrop` and is never set to `invalidpickup`. The assertions could be formulated in other ways, and other assertions could also be added to verify all of the properties specified in [163]. However, for the sake of the case study, these four properties are sufficient to demonstrate the functional verification capabilities of the framework using modeling and model checking.

Because the TASM model of the production cell is finite, it can lend itself quite naturally to the model checking capabilities of the presented framework, using the UPPAAL tool suite. Furthermore, the safety properties that have been expressed as temporal logic formulas can be easily translated to the query language of the UPPAAL model checker.

UPPAAL Model

In order to model check the TASM model for safety assertions, the production cell TASM model leverages the UPPAAL model checker, using the translation approach documented in Appendix C. The UPPAAL model is generated only once and is also used to analyze execution time of the production cell system in Section 8.1.3. Because the timed automata used in UPPAAL do not have hierarchical composition facilities, the TASM main machines need to be "flattened" per the approach described in the proof of Theorem 4.1 and in the proof of Theorem 4.2. The removal of hierarchical composition can lead to exponential growth in the number of states in the "flattened" machine when multiple units of hierarchical composition are used in the same rule. For all machines in the production cell TASM model, except for the *Controller* main machine, the flattening of the machines is tractable because the machines make limited use of hierarchical composition. However, the *Controller* main machine makes heavy use of hierarchical composition within its main rule, and a basic flattening of the main machine yields over one million rules ($3 * 3 * 3 * 3 * 2 * 8 * 3 * 6 * 7 * 3 * 6 * 6 + 1 = 1,765\,969$). Clearly, this approach is not feasible to generate the UPPAAL model. This explosive growth occurs because rule R_1 of machine *Controller* uses 6 sub machines, which, in turn, make use of other sub machines. One way to mitigate the exponential growth is to operate each component in sequence, instead of in parallel. The sequential operation can be achieved by using an extra variable, which orders the operations in sequence. In order to maintain the semantics of the original model, the values of the sensors and actuators are "cached", through "dummy variables", at the beginning of the operation phase and the outputs are "buffered", also through temporary variables, until the end of the operation phase. The *Controller* main machine is modified to use the cached variables in its decisions and to output to the buffered variables. The modified *Controller* main machine is shown in Listing 8.13. The modified main machine contains 9 rules and the "flattened" version of the machine in Listing 8.13 contains 48 rules – 1 rule to flatten rule R_1, 3 rules to flatten rule R_2, 3 rules to flatten rule R_3, 10 rules to flatten rule R_4, 13 rules to flatten rule R_5, 13 rules to flatten rule R_6, 3 rules to flatten rule R_7, 1 rule to flatten rule R_8, and 1 rule for rule R_9. Clearly, this definition of the machine is more manageable. The original model is maintained in order not to affect the modeling because of the translation details. If it can be demonstrated that the Controller model of Listing 8.13 is equivalent to the model of Listing 8.11, then the modified model can be used to generate the UPPAAL timed automata for the controller behavior without loss of semantics.

In order to show equivalence between the machine of Listing 8.13 and the machine of Listing 8.11, two basic principles are invoked. The first one relies on the fact that all controller actions are instantaneous and, hence, occur in the same quantitative

Listing 8.13 Rules of the modified Controller main machine

```
R1: Cache {
  if wait = False and seq = cache then
    CACHE_DATA();
    seq := operate_feed;
}

R2: Feed {
  if wait = False and seq = operate_feed then
    OPERATE_FEED();
    seq := operate_deposit;
}

R3: Deposit {
  if wait = False and seq = operate_deposit then
    OPERATE_DEPOSIT();
    seq := operate_robot;
}

R4: Robot {
  if wait = False and seq = operate_robot then
    OPERATE_ROBOT();
    seq := operate_arma;
}

R5: Robot {
  if wait = False and seq = operate_arma then
    OPERATE_ARM_A();
    seq := operate_armb;
}

R6: Robot {
  if wait = False and seq = operate_armb then
    OPERATE_ARM_B();
    seq := operate_press;
}

R7: Press {
  if wait = False and seq = operate_press then
    OPERATE_PRESS();
    seq := output;
}

R8: Press {
  if wait = False and seq = output then
    OUTPUT();
    seq  := cache;
    wait := True;
}

R9: Wait for a step {
  t := next;

  else then
    wait := False;
}
```

time period. Furthermore, the operations do not depend on one another and can be performed in parallel or in any sequence since they do not share variables. Also, by using the "dummy variables" to cache the state and buffer the output, the decisions at each step are not affected by the actions of other main machines or by the output

Machine	Rules	Flattened Rules
Loader	3	3
Feed	2	2
Robot	2	6
ArmA	5	7
ArmB	5	7
Press	2	2
Deposit	3	3
Controller	2	1, 765 969
Controller'	9	48

Table 8.2: Number of rules for flattened main machines

of each operation. This semantics is equivalent to the semantics of performing each operation in parallel in a single step. The second principle relies on the fact that all changes that modify sensor values involved in the decisions of the controller are achieved through durative actions of other components. The two principles guarantee semantic equivalence because each output of the controller will appear in the sensors after a time delay, hence after a controller step has been performed. The number of rules for each flattened main machine of the production cell TASM model is shown in Table 8.2.

The complete UPPAAL model contains 14 timed automata, including 8 automata for each main machine of the TASM model and 6 automata to enforce the "Else rules" of 6 of the main machines. The timed automata for the *Feed* main machine is shown in Figure 8-2. The safety properties given as temporal logic formulas can be easily translated to UPPAAL 's TCTL query language. The UPPAAL queries corresponding to the safety properties described in the previous section are shown below:

- `A[] (motor_robot == 1) imply (armaext == 1 && armbext == 1)`

- `A[] (armaext == 2) imply (armapos == 1 && armapos == 2)`

- `A[] (error != 2)`

• `A[] (error != 3)`

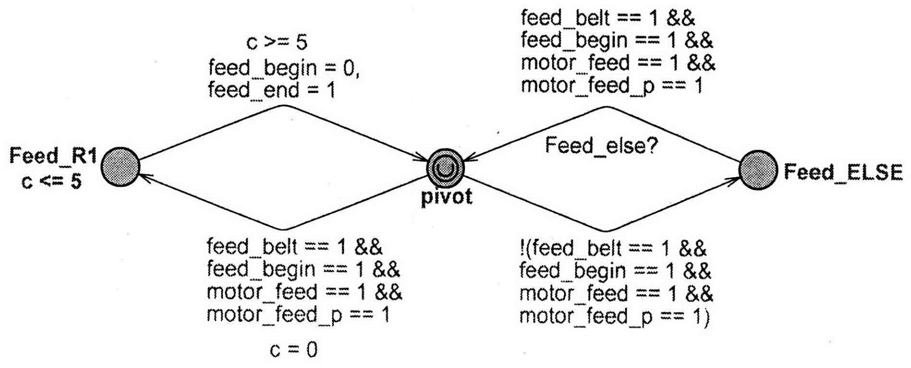


Figure 8-2: Timed automaton for the feed main machine

These properties were verified successfully by running the queries through the UPPAAL verifier. The model can also be queried to verify certain liveness properties. For example, the property "eventually, all blocks loaded into the system get carried out of the system" can be expressed in the query language of UPPAAL . This property can be formulated as the liveness property "`E<> processed_blocks == loaded_blocks`". Other liveness and safety properties can be formulated in a similar fashion, as needed. The UPPAAL model derived in this section is reused in Section 8.1.3 when execution time is analyzed.

Completeness and Consistency

The analysis of completeness and consistency was performed using the approach described in Section 5.1. The results of verifying completeness are shown in Table 8.3. The table shows, for each machine, the number of propositions, the number of clauses, and whether or not the machine is complete. For machines that are trivially complete, the number of propositions and clauses is listed as "N/A". A similar table, Table 8.4, presents the results of verifying the consistency of each machine.

In Table 8.3, the only machine which is not complete is the *Loader* main machine. The machine is not complete because it stops after loading the predefined number of blocks. The counterexample generated by the *SAT* solver is the state where

Name	Propositions	Clauses	Complete
Loader	5	11	No
Feed	N/A	N/A	Yes
Deposit	N/A	N/A	Yes
Press	N/A	N/A	Yes
Robot	2	4	Yes
ArmA	N/A	N/A	Yes
ArmB	N/A	N/A	Yes
Controller	N/A	N/A	Yes
armPosition	N/A	N/A	Yes
rotateClockwise	N/A	N/A	Yes
rotateCounterClockwise	N/A	N/A	Yes
OPERATE_FEED	N/A	N/A	Yes
OPERATE_DEPOSIT	N/A	N/A	Yes
OPERATE_ROBOT	N/A	N/A	Yes
OPERATE_ARM_A	N/A	N/A	Yes
OPERATE_ARM_B	N/A	N/A	Yes
OPERATE_PRESS	N/A	N/A	Yes
PICK_UP_ARM_A	10	23	Yes
PICK_UP_ARM_B	10	23	Yes
DROP_ARM_A	10	23	Yes
DROP_ARM_B	10	23	Yes
ARM_A_FEED	N/A	N/A	Yes
ARM_A_PRESS	N/A	N/A	Yes
ARM_B_DEPOSIT	N/A	N/A	Yes
ARM_B_PRESS	N/A	N/A	Yes
ROBOT_MOTION	N/A	N/A	Yes
ROTATE_ROBOT	N/A	N/A	Yes

Table 8.3: Completeness analysis results for the production cell model

"`loadedblocks >= number, feed_belt = empty`". The machine was designed to stop after the number of loaded blocks exceeds the predefined threshold, so the incompleteness is to be expected. In Table 8.4, main machines *Deposit*, *ArmA*, and *ArmB* are inconsistent. These components model the environment of the controller and hence the lack of consistency uncovers assumptions about the behavior of the environment. For the *Deposit* main machine, the counterexample generated by the *SAT* solver is the state where "`deposit_belt = loaded, deposit_begin = True, motor_deposit = on, motor_deposit_p = negative, deposit_end = True`". In this counterexample, rule R_1 and rule R_2 are both enabled. The assumption about the environment is that `deposit_begin` variable and the `deposit_end` variable cannot be true at the same time. This assumption is congruent with the problem definition

Name	Propositions	Clauses	Consistent
Loader	5	71	Yes
Feed	N/A	N/A	Yes
Deposit	10	15	No
Press	N/A	N/A	Yes
Robot	2	4	Yes
ArmA	10	40	No
ArmB	10	40	No
Controller	N/A	N/A	Yes
armPosition	7	129	Yes
rotateClockwise	N/A	N/A	Yes
rotateCounterClockwise	N/A	N/A	Yes
OPERATE_FEED	6	10	Yes
OPERATE_DEPOSIT	6	10	Yes
OPERATE_ROBOT	N/A	N/A	Yes
OPERATE_ARM_A	6	18	Yes
OPERATE_ARM_B	6	8	Yes
OPERATE_PRESS	6	12	Yes
PICK_UP_ARM_A	10	23	Yes
PICK_UP_ARM_B	10	23	Yes
DROP_ARM_A	10	23	Yes
DROP_ARM_B	10	23	Yes
ARM_A_FEED	12	80	Yes
ARM_A_PRESS	12	80	Yes
ARM_B_DEPOSIT	12	80	Yes
ARM_B_PRESS	12	80	Yes
ROBOT_MOTION	4	8	Yes
ROTATE_ROBOT	16	255	Yes

Table 8.4: Consistency analysis results for the production cell model

which states that the robot will not put a block on the belt if it is already loaded and which can be verified as a safety property. And since the block can be at one end of the belt only and not at both ends simultaneously, only one of the two sensors can be true at any given time. This assumption can be validated against the UPPAAL model by running the query "`A[] !(deposit_end == 1 && deposit_begin == 1)`" to verify that the aforementioned state is not reachable in the model. Similar reasoning can be carried out for the other two inconsistent machines, *ArmA* and *ArmB*. For these machines, the environmental assumption is that the magnet will never be on if an arm is retracted and empty.

8.1.3 Execution Time Analysis

Since the controller model does not contain time annotations, the execution time analysis is not concerned with the performance of the software, but with the performance of the manufacturing system with regards to the controller strategy. For the model, one time property to verify is the amount of time required to process n blocks. This property can be stated over the state variables as the time required to complete a path from a state where "`loaded_blocks = 0, processed_blocks = 0`" to a state where "`loaded_blocks = n, processed_blocks = n`". To demonstrate the approach, n is selected to be 10 blocks. Using the approach described in Section 5.3, an appropriate observer automaton is added to the UPPAAL model. The observer automaton, shown in Figure 8-3, observes the relevant path and measures the time required to complete the path.

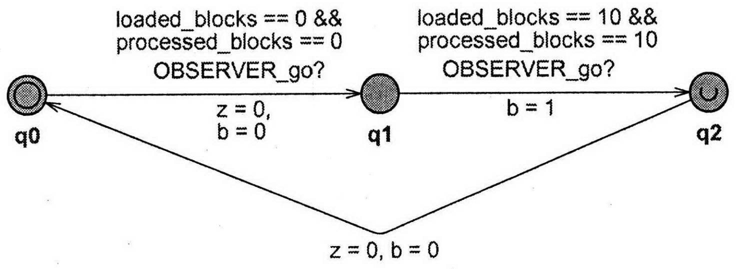


Figure 8-3: Observer automaton to verify the time needed to process 10 blocks

Using the UPPAAL verifier, the time required to process 10 blocks is verified to be 386 time units. Because the timing of the various components is deterministic, 386 time units corresponds to both the best-case execution time and the worst-case execution time, given the controller strategy.

8.1.4 Resource Usage Analysis

The production cell case study contains one resource, power. Power is consumed when the hardware components are operating, for example when the motor of the feed belt is turned on. Analogously to the time annotations, the controller model does not consume any resource and it is justified to assume that the power consumed by the controller is negligible compared to the power consumed by the hardware components. Consequently, the *Controller* machine can be excluded from the iterative algorithm used to converge on the maximum and minimum resource consumption values. According to the algorithm described in Section 5.4, the derivation of minimum and maximum resource consumption uses flattened versions of each main machine. The flattened version of the *Controller* main machine contains the most rules. Removing the controller main machine greatly reduces the complexity of the algorithm. Without the *Controller* main machine, the algorithm described in Section 5.4 iterates through 10584 combinations of rules (3 * 2 * 6 * 7 * 7 * 2 * 3) to find the optimal solution. The maximal solution consumes 6600 units of power, which is obtained by executing rule R_1 of all main machines, in parallel, which consumes the most power, per Table 2.5. As mentioned in Section 5.4, the algorithm used to determine maximum resource consumption can lead to an overapproximation of the consumed resources. The UPPAAL model can be used to determine whether the state which satisfies the maximum resource consumption is reachable. This can be achieved using a simple reachability query over the value of variables in the form of "`E<> (variable values`".

Executing the reachability query for the state which yields the maximal power consumption, as given by the *SAT* solver, reveals that it is not a reachable state. A simple analysis of the suggested state shows that the state is not reachable because the loader cannot be loading a block while the feed belt is on. By iterating through other solutions yielded by the resource consumption verification algorithm, the maximal resource consumption where the satisfying state is reachable yields a resource consumption of 6200 units of power. This state corresponds to both arms being retracted, while the press is stamping a block, the deposit belt is carrying a block, and the feed belt is carrying a block. Summary results are presented in Table 8.5. In the table, the first column shows the value of consumed power. The second column shows, for each main machine, the rule being executed to yield the listed resource consumption. The composite rule names describe which rules are being executed in machines used for hierarchical composition. The third column shows the state which satisfies the parallel execution of the rules. The fourth column indicates whether the state is reachable or not, as verified using the UPPAAL model.

Value	Rules	State	Reachable

6600	Loader: R_1
	Feed: R_1
	Deposit: R_1
	Press: R_1
	ArmA: R_1
	ArmB: R_1
	Robot: $R_1 R_3$
		loaded_blocks < 9, feed_belt = empty,
		feed_belt = loaded, feed_begin = True,
		motor_feed = on, motor_feed_p = positive,
		deposit_belt = loaded, deposit_begin = True,
		motor_feed = on, motor_feed_p = positive,
		motor_press = on, press = loaded,
		press_block = notfinished,
		motor_arma = on, motor_arma_p = positive,
		armaext = retracted, motor_armb = on,
		motor_armb_p = positive, armbext = retracted,
		robot_wait = False, motor_robot = on
			No

6200	Loader: R_3
	Feed: R_1
	Deposit: R_1
	Press: R_1
	ArmA: R_2
	ArmB: R_2
	Robot: $R_1 R_3$
		loaded_blocks < 10, feed_belt = loaded,
		feed_belt = loaded, feed_begin = True,
		motor_feed = on, motor_feed_p = positive,
		deposit_belt = loaded, deposit_begin = True,
		motor_feed = on, motor_feed_p = positive,
		motor_press = on, press = loaded,
		press_block = notfinished,
		motor_arma = on, motor_arma_p = negative,
		armaext = extended, motor_armb = on,
		motor_armb_p = negative, armbext = extended,
		robot_wait = False, motor_robot = on
			Yes

Table 8.5: Resource consumption analysis results for the production cell

For the minimum resource consumption, a trivial case happens after all blocks have been loaded and the robot is at the feed belt, waiting for a block to arrive. In this case, the power consumption is 0 units. The minimum non-zero resource consumption occurs at the beginning of the production process when there are no blocks in the system and the loader is putting the first block on the feed belt. In this case, the power consumption is 200 units.

8.1.5 Test Case Generation

In the production cell case study, the *Controller* main machine is the only main machine describing the behavior of software. The other main machines describe the behavior of hardware components considered part of the environment. Consequently, test cases are generated only for the machines linked to the *Controller* main machine, using the algorithm described in Listing 7.8. The results of the test case generation are shown in Table 8.6. In Table 8.6, the first column provides the machine name, the second column lists the number of test case templates in the test suite template for the machine, and the third column lists the number of test cases from the test suite required for unit testing of the machine. Per the approach described in Chapter 7, the test cases are generated using the rule coverage criterion, explained in Section 7.1.1. Because the test case generation strategy does not require the machines to be "flattened", the generation of test cases is achieved using the *Controller* machine shown in Listing 8.11, not the modified version provided in Listing 8.13. In Table 8.6, the test case results are for unit testing are equal to the number of rules for each machine. Furthermore, the number of test cases in the test suite are the test cases used to achieve both unit testing for the given machine an integration testing for sub machines used in hierarchical composition. The *Controller* main machine contains the most number of test cases, 14, which are used to cover the rules *Controller* main machine and the rules of all the sub machines. The number 14 is obtained through the maximum number of rules for the sub machines, in this case 13, plus 1 more test case used to exercise rule R_2 of the *Controller* main machine.

Machine	**Test Suite**	**Unit Testing**
Controller	14	2
OPERATE_FEED	3	3
OPERATE_DEPOSIT	3	3
OPERATE_ROBOT	10	2
OPERATE_ARM_A	13	3
OPERATE_ARM_B	13	3
OPERATE_PRESS	3	3
ARM_A_FEED	6	6
ARM_A_PRESS	6	6
ARM_B_DEPOSIT	6	6
ARM_B_PRESS	6	6
ROTATE_ROBOT	9	9

Table 8.6: Test case generation results for the production cell model

A sample test case template from the *Controller* main machine test suite is shown in Table 8.7.

Pre State	Post State	Coverage Item
$wait\{False\}$, $motor\_feed\{off\}$, $feed\_begin\{True\}$, $motor\_deposit\{off\}$, $deposit\_begin\{True\}$, $armaext\{extended\}$, $armbext\{extended\}$, $armapos\{intransit\}$, $armbpos\{intransit\}$, $motor\_press\{on\}$, $press\{loaded\}$, $press\_block\{finished\}$	$wait\{True\}$, $motor\_feed\_p\{positive\}$, $motor\_feed\{on\}$, $motor\_deposit\_p\{negative\}$, $motor\_deposit\{on\}$, $motor\_press\{off\}$	$Controller.R_1$, $OPERATE\_FEED.R_1$, $OPERATE\_DEPOSIT.R_1$, $OPERATE\_ROBOT.R_2$, $OPERATE\_ARM\_A.R_3$, $OPERATE\_ARM\_B.R_3$, $OPERATE\_PRESS.R_2$

Table 8.7: Sample test case template from the test suite for the *Controller* main machine

8.1.6 Discussion

The production cell provides a case study of moderate complexity with interesting properties to verify. The TASM model of the production cell contains a model of the controller software, the crux of the model, and simplified models of the hardware components. Having a complete system model enables analysis of system properties, such as resource consumption and the amount of time required to process a fixed number of blocks, e.g., 10 blocks. Furthermore, reachability analysis yielded insight to analyze the reachable states in the controller and in the environment, given the controller strategy. This analysis was performed using safety invariants and using errors embedded in the model. The completeness and consistency analysis uncovered interesting environmental assumptions that could be validated using the UPPAAL model, such as the impossibility that the `feed_begin` and `feed_end` sensors be true simultaneously. The completeness and consistency analysis proved to be a valuable aid in building and debugging the TASM model. While the completeness and consistency results aren't incredibly insightful once the model is completed, the capabilities were useful during development.

The translation from TASM to UPPAAL uncovered an important limitation in the translation algorithm with respect to the hierarchical composition mechanisms of the TASM language. Because UPPAAL does not have hierarchical composition facilities, the construction of a "flattened" machine is a necessary step of the translation. The number of rules of the "flattened" machine grows exponentially with the number of sub machines used in parallel for hierarchical composition, for a given rule. Nevertheless, with minor modeling modifications, an equivalent TASM model was derived, which yielded a scalable translation. The primary purpose of the production cell case study is to demonstrate the modeling and analysis capabilities of the proposed framework. As such, the case study proved useful since it provides a model of moderate size and demonstrates the analysis of functional properties, execution time, and resource consumption.

The test case generation for the production cell case study was fairly straightforward since the *Controller* main machine and its associated sub machines do not use real or integer variables. Furthermore, when hierarchical composition is utilized, the composed machines do not share variables, hereby reducing the complexity of combining the test suites of different machines. It is interesting that the test case generation strategy proved scalable, even through the heavy use of hierarchical composition in the *Controller* main machine. This situation would not have been possible using model checking approaches, unless the model was altered as described in Listing 8.13.

8.2 Electronic Throttle Controller: High Level Model

The Electronic Throttle Controller (ETC) case study is based on a Simulink model [167] developed by Griffiths et al. and by an industrial automotive manufacturer [111]. The case study is described in details in Section 2.8.2. The case study is used to illustrate the modeling capabilities of the TASM language, to exercise the analysis features of the framework, to illustrate the bi-directional traceability approach, and to demonstrate the test case generation capabilities, including regression testing. The case study is presented in three different sections. The first version of the model, described in this section, describes a high level model of the ETC for the mode switching logic of the controller and for the calculation of the desired current. The following section, Section 8.3, presents a tasking model and a scheduler which are used to implement the ETC functionality. Finally, Section 8.4 presents the combination of the tasking model with the high level model presented in this section, and adds resource consumption to the model.

As a reminder, the ETC operates based on "modes", which are set by the controller based on environmental factors such as vehicle parameters, driver inputs, and climate conditions. The operation of the ETC is divided into major modes of operation and minor modes of operation. During nominal operation, the major modes of the controller are grouped into "driving modes" and "limiting modes". The limiting modes, defined as undesirable vehicle conditions that need to be remedied by the controller, take precedence over driving modes. Limiting modes are divided in minor modes, namely "traction control", where the wheels rotate with too little friction, and "revolution control", where the engine operates over a threshold rotation per minute. The driving modes are also divided into minor modes of operation, namely "human control", where the driver dictates the behavior of the controller, and "cruise control", where the behavior of the controller is determined using set parameters. The different modes are shown in Figure 8-4, adapted from [111], represented visually as a statechart variant [120]. The "XOR" label indicates mutual exclusion between modes and the "AND" label indicates parallel composition of modes.

As a reminder, the ETC uses the major and minor modes of operation to calculate the output of the ETC, the desired current, which is output to the throttle. The desired current dictates the throttle angle and controls how much air enters the engine and, consequently, on much torque and RPM the engine produces. The high level requirements governing the logic for determining the modes of operation and the associated calculation of the desired current can be summarized as follows:

- **Req 1.1**: The controller shall operate in two major modes, the *driving* mode and the *limiting* mode
  - **Req 1.1.1**: The driving mode shall be active when the *limiting condition*

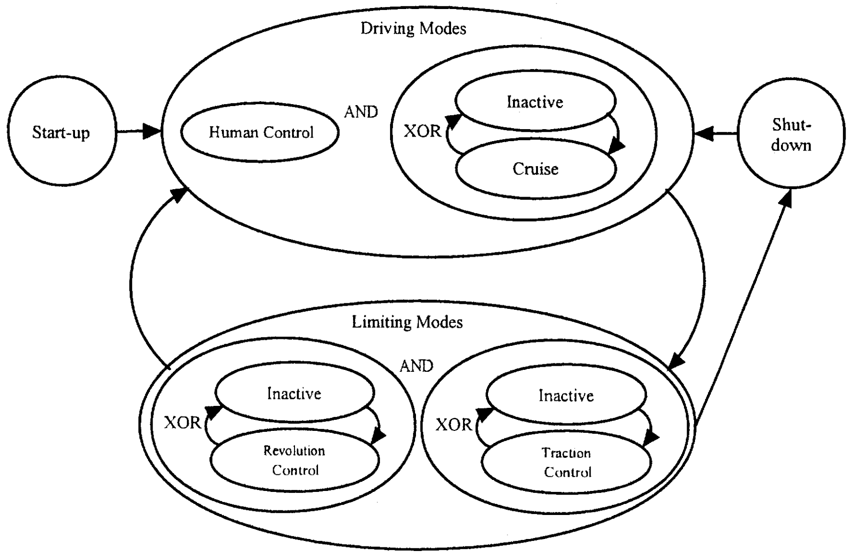


Figure 8-4: ETC modes

is False.

  - **Req 1.1.2**: The limiting mode shall be active when the *limiting condition* is True.

  - **Req 1.1.3**: The limiting mode and the driving mode shall be mutually exclusive and collectively exhaustive.

- **Req 1.2**: The limiting mode shall be divided into over revolution mode and traction control mode

  - **Req 1.2.1**: The over revolution mode shall be active when the engine RPM is over 6000 rotations per minute

  - **Req 1.2.2**: The traction limiting mode shall be active when the engine torque is over 110 kPa

- **Req 1.3**: The driving mode shall be divided into human mode and cruise control mode

  - **Req 1.3.1**: The cruise control mode shall be active when the gear is in drive, the vehicle speed is over 30 miles per hour, the break pedal is depressed, and the cruise switch is on

  - **Req 1.3.2**: In all other conditions, the control mode shall be inactive

- **Req 1.4**: When calculating the desired current, the limiting modes shall take precedence over the driving modes

  - **1.4.1**: When the limiting mode and the driving modes are active, the desired current is calculated using the limiting mode

    * **1.4.1.1**: When the over revolution limiting mode and the traction limiting mode are active, the desired current is calculated using the minimum value of the desired current calculated using these two modes separately

    * **1.4.1.2**: When only one of the limiting modes is active, this mode is used to calculate the desired current

  - **1.4.2**: When the limiting mode is inactive, the desired current is calculated using the driving mode

    * **1.4.2.1**: When both the human driving mode and the cruise control driving modes are active, the desired current is calculated using the maximum value of the desired current calculated using these two modes separately

    * **1.4.2.2**: When only one of these modes is active, this mode is used to calculate the desired current

8.2.1 Model

The Simulink model of the ETC is adapted into the TASM language by modeling the control of the desired current as a main machine. This main machine, called *CONTROLLER*, implements the requirements described above by performing 5 main steps: reading the state of the environment (vehicle, driver inputs, and climate) through sensors, setting the mode of the controller (major and minor), calculating the output current, and monitoring the health of the system. When the car is turned on, this sequence of operations executes indefinitely in a loop, until the car is turned off. The rules of the *CONTROLLER* main machine are shown in Listing 8.14.

Listing 8.14 Rules of the *CONTROLLER* main machine

```
R1: Controller loop when nominal {
  if control_mode = sample then
    SAMPLE_STATE();
    control_mode := mode_set;
}

R2: Controller loop to set major mode {
  if control_mode = mode_set_major then
    SET_MAJOR_MODE();
    control_mode := mode_set_minor;
}

R3: Controller loop to set minor mode {
  if control_mode = mode_set_minor then
    SET_MINOR_MODE();
    control_mode := output;
}

R4: Controller loop to output current {
  if control_mode = output then
    CALCULATE_OUTPUT();
    control_mode := health;
}

R5: Controller loop to find failure {
  if control_mode = health then
    MONITOR_HEALTH();
    control_mode := sample;
}
```

Per the requirements, the mode of operation of the controller is set using a major mode and a minor mode. The steps of the actions of the controller are captured through the `Control_mode` data type. Since this version of the ETC model is a high level model, the output current is abstracted through a user-defined type called `Desired_Current`, as shown in Listing 8.15. The current is abstracted using the type because it eases verification and makes explicit what actual current is being output based on the logic. This simplification is justified for a high level model, and can later be refined when the calculation of the current is implemented using a specification type of controller.

The `Mode` datatype is used to set the major mode of operation. The `Binary_Mode` datatype is used to set the cruise, limiting, over revolution, and over torque minor

Listing 8.15 User-defined types of the ETC high level model

```
Binary_Mode     := {active, inactive};
Binary_Status   := {on, off};
Health_Status   := {nominal, fault_detected};
Mode            := {off, startup, shutdown, driving, limiting, faulty};
Gear_Status     := {park, drive};
Control_Mode    := {sample, mode_set_major, mode_set_minor, output, health};
Desired_Current := {none_c, human_c, cruise_c, traction_c, rev_c, min_limiting_c,
                     max_driving_c, fault_c, error_c};
Simulation_Mode := {begin_s, drive_s, random_s, stop_s};
```

modes. The high level ETC model does not contain time or resource annotations since the model describes high level behavior. The model makes heavy use of hierarchical composition to structure the logic for mode switching and for the logic to calculate the desired current. The `Simulation_Mode` datatype is used to express various simulation scenarios, as explained in the functional analysis section, Section 8.2.2..

Complete Model

The complete TASM model contains 3 main machine, 13 function machines, and 10 sub machines. The complete electronic throttle controller high level model is documented in Appendix E, Section E.1, where the list of all machines is shown in Table E.1.

8.2.2 Functional Analysis

The functional analysis of the high level model concerns the correct transitions between modes and the correct calculation of the desired current based on the mode of operation. The analysis of the model is achieved through both simulation scenarios, model checking, and verification of completeness and consistency.

Scenario Modeling

The electronic throttle controller reacts to changes in the state of the vehicle (vehicle speed, wheel traction, etc.) and operator inputs (gas pedal angle, cruise control switch, ignition, gear position, etc.). In order to exercise the various modes of the

ETC, different simulation scenarios were devised. For example, the throttle controller always begins in the "off" mode, until ignition is turned on. Because the throttle controller will do nothing until the ignition is turned on, the functionality of the ETC can be exercised by either selecting different initial conditions e.g., starting the ETC in "driving" mode, or by modeling environment behavior to trigger the different modes. For example, nominal operation of the driver would turn on the ignition, put the car in drive, and start driving by pressing the gas pedal. The cruise control mode can only be initiated by the driver, by setting the cruise switch to the on position, when the other conditions of the vehicle are met. Traction mode and engine revolution mode are triggered through a combination of driver input and environmental conditions such as a slippery or uneven road surface.

In order to exercise the behavior of the controller, a main machine modeling driver behavior and a main machine modeling the car behavior have been designed. The first machine, called *DRIVER*, simply performs nominal initiation actions by turning on the ignition, setting the car gear in drive, and activating the gas pedal. Once the driver has transitioned the car into nominal driving mode, the driver can arbitrarily press the inputs – the cruise control switch, the break pedal, and the gas pedal. The second machine, called *VEHICLE*, arbitrarily varies the vehicle speed, engine speed, and vehicle traction. The detailed dynamics of the vehicle and the driver transfer function are not modeled in details, but are encoded into hardcoded values. As such, the behavior of the driver and the behavior of the vehicle are not directly linked. For example, if the driver presses the gas pedal, it won't necessarily directly translate to an increase in the speed of the vehicle. While this lack of causality seems nonintuitive, it leads to a much more simplified model. Furthermore, the behavior of the driver and the behavior of the vehicle, as modeled in the two machines, provide an overapproximation of the true behavior of the vehicle and driver. For example, it is possible for the driver to press the gas pedal and for the vehicle to decrease its velocity. This behavior might seem counterintuitive, but it captures a richer set of scenarios and makes few assumptions about the behavior of both the driver and the vehicle. The only restrictions on the reachable states of both the driver and the vehicle follow widely accepted automotive principles such as the car gear being in "drive" can only occur if the ignition is "on" and the ignition can be turned off only if the car gear is in "park". Furthermore, the car gear can only be in "park" if the vehicle speed is 0.

The *DRIVER* main machine will arbitrarily affect the driver inputs. Three sample rules of the DRIVER main machine are shown in Listing 8.16. These rules will non-deterministically change the status of the cruise switch with nominal speed, rule R_4, and with speed that is too low for the cruise control to take effect in the ETC, through rule R_3. The complete DRIVER main machine is documented in Listing E.4 and in Listing E.5.

Listing 8.16 Two rules of the *DRIVER* main machine

```
R3: Turn on cruise, slow speed
{
  if driver_s = random_s and cruise_switch = off then
    cruise_switch := on;
    vehicle_speed := 10;
}

R4: Turn on cruise, normal speed
{
  if driver_s = random_s and cruise_switch = off then
    cruise_switch := on;
    vehicle_speed := 30;
}

R5: Turn off cruise
{
  if driver_s = random_s and cruise_switch = on then
    cruise_switch := off;
}
```

The *VEHICLE* main machine will arbitrarily affect the state of the vehicle and will wait for the proper action by the controller to return the vehicle to a nominal state. The combination of the *DRIVER* main machine and the *VEHICLE* main machine exercise the full behavior of the controller. Two sample rules of the *VEHICLE* main machine are shown in Listing 8.17. These rules will non-deterministically change the engine speed and the vehicle torque over the desired threshold, through rule R_4, and wait until the correct desired current is output to return them to nominal values, as achieved through rule R_7. The complete *VEHICLE* main machine is documented in Listing E.6 and in Listing E.7.

Listing 8.17 Two rules of the *VEHICLE* main machine

```
R4: Randomly change Both
{
  if driver_s = random_s and vehicle_over_rev_s = False and
     vehicle_over_tor_s = False then
    engine_speed       := MAX_ENGINE_SPEED + 1;
    vehicle_over_rev_s := True;
    vehicle_torque     := MAX_TORQUE + 1;
    vehicle_over_tor_s := True;
}

R7: Randomly change both, correct
{
  if driver_s = random_s and vehicle_over_rev_s = True and
     vehicle_over_tor_s = True and desired_current = min_limiting_c then
    vehicle_torque     := MAX_TORQUE;
    vehicle_over_tor_s := False;
    engine_speed       := MAX_ENGINE_SPEED;
    vehicle_over_rev_s := False;
}
```

Safety and Liveness Properties

Verifying the safety properties of the controller involves asserting that certain states are not reachable while certain states must be reached after a certain condition is met. Since the controller operates in steps, it is important to formulate the state conditions over the "cached" state in order to avoid writing queries over the state that could change during the controller sequence of steps. The safety properties to be verified fall into two distinct categories – verifying that the correct mode is set depending on environmental conditions and verifying that the correct output current is set depending on the mode of operation.

In the first category of properties, verifying the that the limiting mode is correctly set involves checking that, whenever the engine revolution is over the maximum threshold or the vehicle torque is over the maximum threshold, the controller is in the limiting mode. These two properties can be easily formulated as temporal logic formulas:

- *A G (c_engine_speed > MAX_ENGINE_SPEED ∧ control_mode = output) → (controller_mode = limiting)*

- *A G (c_vehicle_torque > MAX_TORQUE ∧ control_mode = output) → (controller_mode = limiting)*

These two properties are stated over the "cached" values of the state `c_engine_speed` and `c_vehicle_torque` to avoid cases involving non-deterministic state changes during controller operation. Furthermore, the `control_mode` variable is added to the query to ensure that the mode is correctly set *after* the controller has finished setting the mode. In the temporal logic queries, for the properties of interest, the `control_mode` variable should set to `output` to denote that the mode has already been set. Similar queries can be formulated for the driving mode and for the other minor modes such as the cruise control mode being set when the cruise condition is enabled. The second category of properties involves verifying that the correct desired current is output based on the controller mode. The correct desired current is defined according to the requirements listed in Section 8.2. Some basic properties can be stated about the desired current to ensure that minimal guarantees exist. For example, whenever the controller mode is driving, the desired current is either the human current, the cruise control current, or the minimum value of the two currents. In temporal logic, this property can be stated as:

- *A G (controller_mode = driving ∧ control_mode = health) → (desired_current = human_c ∨ desired_current = cruise_c ∨ desired_current = max_driving_c)*

The query uses the `control_mode` variable to ensure that the desired current is correctly calculated *after* the controller has finished calculating the current. This is achieved by ensuring that the `control_mode` variable is set to `health`, which occurs only after the desired current has been calculated. The query concerning the desired current also illustrate the benefits of abstracting away the actual numerical value of the current being output. If a real variable were output, it would be difficult to determine exactly what numerical value the current should be without taking into account the actual speed and dynamics of the vehicle. A similar query can be formulated for the limiting major mode. More complex properties can be verified by querying based on both major and minor modes and ensuring that the current calculation logic is correct with respect to the requirements. For example, the limiting mode takes precedence over the driving mode and if both the over revolution mode and the traction mode are active, the minimum current of both limiting modes is used as the desired current output. This property can be formulated as:

- $A\ G\ (controller\_mode = limiting \wedge cruise\_mode = active \wedge rev\_limiting\_mode = active \wedge traction\_mode = active \wedge control\_mode = health) \rightarrow (desired\_current = min\_limiting\_c)$

Similar properties can be formulated for the other combination of modes. After establishing that the modes are correctly set based on environmental conditions and establishing that the output current is correctly calculated based on the controller mode, it naturally follows that the ETC behaves correctly according to the requirements.

UPPAAL Model

In order to verify the TASM model for the aforementioned safety assertions, the ETC TASM model leverages the UPPAAL model checker, using the translation approach documented in Appendix C. Because the timed automata used in UPPAAL do not have hierarchical composition facilities, the TASM main machines need to be "flattened" per the approach used in the proof of Theorem 4.1 and in the proof of Theorem 4.2. The removal of hierarchical composition leads to exponential growth in the number of states in the "flattened" machine where multiple units of hierarchical composition happen in parallel. In the ETC model, only the *CONTROLLER*

Machine	Rules	Flattened Rules
CONTROLLER	5	33
DRIVER	11	11
VEHICLE	9	9

Table 8.8: Number of rules for flattened main machines for the high level ETC model

main machine uses hierarchical composition and is the only machine that needs to be flattened. In the definition of the *CONTROLLER* main machine, the invocation of the *SET_MINOR_MODE_WORK* sub machine is the only place where hierarchical composition is performed in parallel using multiple sub machines. Consequently, this is the only place in the model that leads to exponential growth in the number of rules of the flattened machine. However, this growth is tractable since it leads to only 8 rules (2 * 2 * 2). The number of rules for each flattened main machine of the TASM model is shown in Table 8.8.

The complete UPPAAL model contains 5 timed automata, including 3 automata for each main machine of the TASM model and 2 automata to enforce the "Else rules" of the *DRIVER* main machine and of the *VEHICLE* main machine. The safety properties given as temporal logic formulas can be easily translated to UPPAAL 's TCTL query language. The UPPAAL queries corresponding to the safety properties stated above are shown below, in the order in which they were introduced:

- `A[] (c_engine_speed > MAX_ENGINE_SPEED && control_mode == 4 && system_health == 1) imply (controller_mode == 5)`

- `A[] (c_vehicle_torque > MAX_TORQUE && control_mode == 4 && system_health == 1) imply (controller_mode == 5)`

- `A[] (controller_mode == 4 && control_mode == 5) imply (desired_current == 2 || desired_current == 3 || desired_current == 7)`

- `A[] (controller_mode == 5 && cruise_mode == 1 && rev_limiting_mode == 1 && traction_mode == 1 && control_mode == 5) imply (desired_current == 6)`

These properties were successfully verified by running the queries through the UPPAAL verifier. The first two properties differ slightly from the original stated assertions because if a fault is present, the controller mode will be set to the faulty mode instead of being set to the limiting mode. The detection of a fault takes precedence over all the other modes of operation. The extended query adds a condition to state that the limiting mode is set when one of the two limiting conditions are true and the system health is nominal, through the `system_health` variable. The model can also be queried to verify certain liveness properties, to ensure that the model behaves correctly. For example, the property "eventually, the car is in cruise control" can be verified. This property can be formulated as the liveness property "`E<> cruise_mode == 1`". Other liveness and safety properties can be formulated in a similar fashion, as needed.

Completeness and Consistency

The analysis of completeness and consistency was performed using the approach described in Section 5.1. The results of verifying completeness are shown in Table 8.9. The table shows, for each machine, the number of propositions, the number of clauses, and whether or not the machine is complete. For machines that are trivially complete, the number of propositions and clauses is listed as "N/A". A similar table, Table 8.10, presents the results of verifying the consistency of each machine.

In Table 8.9, all machines are complete. In Table 8.10, machines *DRIVER* and *VEHICLE* are not consistent, as expected. These components model the environment of the controller and hence the lack of consistency uncovers assumptions about the behavior of the environment. For the *DRIVER* machine, one of the generated counterexamples by the *SAT* solver is the state where "`driver_s = random, cruise_switch`

Name	Propositions	Clauses	Complete
CONTROLLER	5	32	Yes
DRIVER	N/A	N/A	Yes
VEHICLE	N/A	N/A	Yes
Cruise	N/A	N/A	Yes
Cruise_Mode	N/A	N/A	Yes
Cruise_Throttle_C	N/A	N/A	Yes
Driver_Throttle_C	N/A	N/A	Yes
Driving_Throttle_C	N/A	N/A	Yes
Fault	N/A	N/A	Yes
Limiting_Throttle_C	N/A	N/A	Yes
Over_Rev	N/A	N/A	Yes
Over_Rev_Mode	N/A	N/A	Yes
Over_Rev_Throttle_C	N/A	N/A	Yes
Over_Torque	N/A	N/A	Yes
Over_Torque_Mode	N/A	N/A	Yes
Over_Torque_Throttle_C	N/A	N/A	Yes
CALCULATE_OUTPUT	6	64	Yes
DO_SHUTDOWN	N/A	N/A	Yes
DO_STARTUP	N/A	N/A	Yes
HANDLE_FAULT	N/A	N/A	Yes
MONITOR_HEALTH	N/A	N/A	Yes
SAMPLE_STATE	N/A	N/A	Yes
SET_MAJOR_MODE	N/A	N/A	Yes
SET_MAJOR_MODE_WORK	N/A	N/A	Yes
SET_MINOR_MODE	N/A	N/A	Yes
SET_MINOR_MODE_WORK	N/A	N/A	Yes

Table 8.9: Completeness analysis results for the ETC high level model

`= off`". In this state, rule R_3 and rule R_4 are both enabled. This behavior is expected since it was modeled this way, to simulate driver behavior where the cruise switch is set to on and the driver speeds up over the cruise speed threshold, and to simulate the driver behavior where the cruise switch is set to on and the driver slows down under the cruise speed threshold. For the *VEHICLE* machine, the generated counterexample is the state where "`driver_s = random, vehicle_over_rev_s = False, vehicle_over_tor_s = False`". In this state, rule R_1 and rule R_2 are both enabled. In a similar fashion, this behavior is expected since these two rules were designed to model vehicle behavior which stays nominal, and vehicle behavior which arbitrarily transitions to an engine RPM over the maximum threshold.

Since the high level ETC model does not contain time or resource annotations, this version of the model does not contain analysis results for execution time and for

Name	Propositions	Clauses	Consistent
CONTROLLER	5	32	Yes
DRIVER	18	88	No
VEHICLE	19	529	No
Cruise	N/A	N/A	Yes
Cruise_Mode	N/A	N/A	Yes
Cruise_Throttle_C	N/A	N/A	Yes
Driver_Throttle_C	N/A	N/A	Yes
Driving_Throttle_C	5	11	Yes
Fault	N/A	N/A	Yes
Limiting_Throttle_C	4	7	Yes
Over_Rev	N/A	N/A	Yes
Over_Rev_Mode	N/A	N/A	Yes
Over_Rev_Throttle_C	N/A	N/A	Yes
Over_Torque	N/A	N/A	Yes
Over_Torque_Mode	N/A	N/A	Yes
Over_Torque_Throttle_C	N/A	N/A	Yes
CALCULATE_OUTPUT	6	64	Yes
DO_SHUTDOWN	N/A	N/A	Yes
DO_STARTUP	N/A	N/A	Yes
HANDLE_FAULT	N/A	N/A	Yes
MONITOR_HEALTH	N/A	N/A	Yes
SAMPLE_STATE	N/A	N/A	Yes
SET_MAJOR_MODE	8	63	Yes
SET_MAJOR_MODE_WORK	16	78	Yes
SET_MINOR_MODE	N/A	N/A	Yes
SET_MINOR_MODE_WORK	N/A	N/A	Yes

Table 8.10: Consistency analysis results for the ETC high level model

resource consumption.

8.2.3 Test Case Generation

In the ETC case study, all the machines describe the behavior of the software except for the *DRIVER* and the *VEHICLE* main machines which describe environmental behavior. Consequently, test suites are generated for all the machines except for the *DRIVER* and *VEHICLE* machines, using the algorithm described in Listing 7.8. The results of the test case generation are shown in Table 8.7. In Table 8.7, the first column provides the machine name, the second column lists the number of test case templates in the test suite template for the machine, and the third column lists the number of test cases from the test suite required for unit testing of the machine. Per the approach described in Chapter 7, the test cases are generated using the rule coverage criterion, explained in Section 7.1.1. Because the ETC high level model contains multiple function machines, some transformations need to be applied to the model in order to generate test cases, as explained in Section 7.4. For example, when function machines are used in a rule guard, the guard needs to be rewritten so that the function machine is removed, per the approach described in the proof of Theorem 4.1. Furthermore, function machines used in effect expressions are converted to sub machines, as described in Section 7.4. The *CONTROLLER* main machine contains the most test cases in its test suite, 33 in total, because it is the most complex machine, since it uses hierarchical composition. Because the machine uses hierarchical composition in many of its rules, the total number of rules required to cover the rules of the *CONTROLLER* main machine and the rules of the sub machines it uses will be approximately the sum of the number of rules of all the machines in the model. The total number of rules is slightly lower than the total summation because certain rules can be covered in parallel during the invocation of the *SET_MINOR_MODE_WORK* sub machine, which uses multiple units of hierarchical composition in its rules.

A sample test case template from the *CONTROLLER* main machine test suite is shown in Table 8.12.

8.2.4 Discussion

Adapting the Simulink ETC model into the TASM language enabled the verification of consistency and completeness for the mode switching logic and desired current calculation logic. Determining these properties was helpful during the early modeling stages to ensure that no cases were missed and that no cases were conflicting. Some of this analysis uncovered inconsistencies and incompleteness in the Simulink model. For example, the cruise control throttle current calculation was unreachable in the Simulink model. Furthermore, the Simulink model does contain the modeling of non-deterministic driver behavior to set the cruise switch or to activate the break pedal. The Simulink model contains driver behavior for gas pedal input only, which is modeled as a deterministic scenario using a hardcoded transfer function.

The completeness and consistency verification was established in isolation, by

Machine	Test Suite	Unit Testing
CONTROLLER	33	5
Cruise	2	2
Cruise_Mode	3	3
Cruise_Throttle_C	1	1
Driver_Throttle_C	1	1
Driving_Throttle_C	3	3
Fault	2	2
Limiting_Throttle_C	6	4
Over_Rev	2	2
Over_Rev_Mode	3	2
Over_Rev_Throttle_C	1	1
Over_Torque	2	2
Over_Torque_Mode	3	2
Over_Torque_Throttle_C	1	1
CALCULATE_OUTPUT	15	6
DO_SHUTDOWN	2	2
DO_STARTUP	2	2
HANDLE_FAULT	2	2
MONITOR_HEALTH	3	2
SAMPLE_STATE	1	1
SET_MAJOR_MODE	11	3
SET_MAJOR_MODE_WORK	9	7
SET_MINOR_MODE	3	2
SET_MINOR_MODE_WORK	3	1

Table 8.11: Test case generation results for the ETC high level model

Pre State	Post State	Coverage Item
$control\_mode\{health\}$, $fault\{False\}$	$control\_mode\{sample\}$, $system\_health\{nominal\}$	$CONTROLLER.R_5$, $MONITOR\_HEALTH.R_2$

Table 8.12: Sample test case template from the test suite for the *CONTROLLER* main machine

verifying the properties on a machine-by-machine basis. The hierarchical composition of the TASM language and toolset ensures that systems can be designed bottom-up using pre-verified auxiliary machines that are known to be complete and consistent. The completeness and consistency results will be reused in the following sections, where the ETC model developed in this section is extended and implemented using a set of tasks and a scheduler.

By translating the TASM model to UPPAAL , safety properties were verified to ensure that the controller behaves correctly with respect to the requirements. The original verification of the controller also uncovered corner cases that the Simulink model did not cover, such as the injection of a fault overriding the limiting and driving controller modes. This case study provided a good example of an embedded controller and how a high level model can be analyzed using the proposed framework. The refined version of this model, presented in the following sections builds on the model and analysis of this section and illustrates other features of the presented framework, including the bi-directional traceability capabilities and the generation of test cases for regression testing.

8.3 Electronic Throttle Controller: Tasking Model

The high level version of the electronic throttle controller, presented in Section 8.2, focused on the logic used to set the mode of operation of the controller and focused on the logic used to calculate the desired current. In this section, the tasking model and the scheduler used to implement the ETC are presented and analyzed using the framework. The tasking model and scheduler presented in this section are later used, in Section 8.4, to implement the mode setting logic and desired current calculating functionality of the ETC. The model presented in this section introduces the time-dependent behavior of the throttle controller.

In the Simulink model described in [111], the functionality of the throttle controller is implemented using three tasks – a manager task, a monitor task, and a servo task. The manager task is responsible for setting the major and minor mode of the controller. The monitor task is responsible for detecting failures and monitoring the health of the system. The servo task is responsible for calculating and outputting the desired current. The three tasks operate at different frequencies, based on the performance requirements of the ETC. The servo task is responsible for the closed-loop control of the ETC and must operate at a rate of 300 Hz. The manager task must operate at a rate of 100 Hz. The monitor task is less critical with an operation frequency of 30 Hz being acceptable. The tasks are driven by a scheduler which operates at a clock speed of 1 kHz. The performance requirements have been obtained from [111].

In order to model the tasks and to perform analysis of the scheduling strategy, the Best-Case Execution Time (BCET) and Worst-Case Execution Time (WCET) of each task must be estimated. These calculations have been performed in past research in [48] by aggregating the operations from the Simulink model and calculating the number of clock cycles required for each operation on a PowerPC 405. The number of clock cycles for each operation is shown in Table 8.13.

Operation	Execution Time
Branch	1-3
Division	120
Multiplication	20
Addition	4-8

Table 8.13: Execution time for floating point operations for a PowerPC 405 (in clock cycles)

For each task, the number of operations is compiled and the resulting BCET and WCET can be estimated using the PowerPC 405 processor operating at 10MHz. The resulting values of BCET and WCET are shown in Table 8.14.

Task	Manager	Monitor	Servo
#Branches	2-5	4	6
#Divisions	0	0	1
#Multiplications	0	80	30
#Additions	0-1	24	32
BCET (cycles)	2	1700	854
WCET (cycles)	23	1804	994
BCET (ms)	0.0002	0.17	0.0854
WCET (ms)	0.0023	0.1804	0.0994
Period (ms)	10	30	3

Table 8.14: Timing properties for the ETC tasks

These timing estimates are a bit coarse and rely on the details of the implementation. While this level of analysis might seem inappropriate for the level of modeling performed in this section, the coarseness of the analysis yields realistic upper and lower bounds on the timing of the tasks, bounds which can readily be used for modeling at the task level. Furthermore, for BCET execution, certain properties such as pipelining, instruction caching, and data caching were ignored. Sophisticated tools and algorithms for execution time analysis exist, as documented in [89], and can later be used when the functionality is implemented to validate the early estimates. Furthermore, the bi-directional strategy can be leveraged to track the assumptions about the BCET and WCET of each task. Afterall, the goal of this model is not to derive precise metrics, but to demonstrate the capabilities of the TASM language and associated framework. Consequently, it is justifiable to round the execution time estimates into safe and tight bounds. In order to discuss the timing data in terms of integers, μs are used. For the manager task, the execution time bounds can be rounded to [0, 5] μs. For the monitor task, the execution time bounds can be rounded to [100, 200] μs. For the servo task, the execution time bounds can be rounded to [70, 100] μs. The scheduling algorithm used in the Simulink model does not follow the optimal and widely used Rate Monotonic Analysis (RMA) [58] scheduling algorithm. Instead, the Simulink model gives precedence to the manager task first, followed by the monitor task, followed by the servo task. The priority scheme is statically assigned and hardcoded in Stateflow. Furthermore, the scheduler is non-preemptive, meaning that tasks cannot be interrupted once they have begun execution.

Since the execution times of each task are much smaller than the periods of each task, it is obvious with simple analysis to observe that the set of tasks is schedulable. Furthermore, given the priority scheme, the major cycle, that is, the amount of time required for all tasks to complete and start repeating the pattern, is 30 ms; the major cycle is clearly identified since the combined worst-case execution time of all tasks, 305 μs, fits inside of the period of the task with the fastest rate, the servo task, whose period is 3 ms. Furthermore, taking into account the resolution of the scheduler, 1 ms, the worst-case execution time for each individual tasks will fit inside the 1 ms blocks meaning that the maximum delay before the beginning of the execution of a task will be 2 ms. Since the worst-case execution time of the servo task is 100 μs, taking into account the potential 2 μs block delays, the worst-case completion time for the servo task would be 2100 μs, which still falls under the deadline of 3 ms.

8.3.1 Model

The tasking structure and the scheduler are modeled in the TASM language to perform various forms of analysis, including schedulability analysis. The 1 ms clock is modeled as a main machine, named *SCHEDULER* which constantly performs steps in 1 ms increments. The tasks are modeled as a single main machine, named *TASKS*, to reflect the single processor implementation. Tasks can be in 4 possible states – waiting, released, executing, and finished. The waiting state denotes a task that is waiting for the next period to begin its execution. The released state denotes a task that is ready to execute. The executing state denotes a task that is currently executing. The finished state denotes a task that has finished execution for the current release. The *TASKS* main machine is purely reactive and will perform steps if one of the tasks is in the executing state. Otherwise, the machine waits for a task to be granted execution. The main machine representing the task execution is given in Listing 8.18. In Listing 8.18, the time annotations correspond to the BCET and WCET estimates derived in the previous subsection.

The states of the tasks are manipulated by the scheduler, which is also modeled as a main machine. The scheduler follows the ticks of the 1 ms clock, releases tasks when the periods expire, and sends tasks to execute on the processor based on the priority scheme. There are no stoppage conditions for the model. The complete model of the scheduler and tasks contains 3 main machines, 1 function machine, and 10 sub machines. The complete scheduler model is documented in Appendix E, Section E.2, where the complete list of all machines is given in Table E.2.

8.3.2 Functional Analysis

Modeling the ETC tasking model in the TASM language enables analysis through model checking by translating the TASM model to UPPAAL . The properties that can be verified within UPPAAL include schedulability, that is, the absence of missed

Listing 8.18 *TASKS* main machine

```
R1: Execute manager {
  t := [0, 5];

  if manager_s = executing then
    manager_s := finished;
}

R2: Execute monitor {
  t := [100, 200];

  if monitor_s = executing then
    monitor_s := finished;
}

R3: Execute servo {
  t := [70, 100];

  if servo_s = executing then
    servo_s := finished;
}

R4: Else, do nothing, wait for an event {
  t := next;

  else then
    skip;
}
```

Machine	Rules	Flattened Rules
CLOCK	2	16
SCHEDULER	4	28
TASKS	4	4

Table 8.15: Number of rules for flattened main machines

deadlines, and safety and liveness properties of the scheduler logic.

UPPAAL Model

In order to model check the TASM model for safety assertions and schedulability properties, the ETC TASM model leverages the UPPAAL model checker, using the translation approach documented in Appendix C. Because the timed automata used in UPPAAL do not have hierarchical composition facilities, the TASM main machines need to be "flattened" per the approach in the proof of Theorem 4.1 and in the proof of Theorem 4.2. The removal of hierarchical composition leads to exponential growth in the number of states in the "flattened" machine where multiple units of hierarchical composition happen in parallel. In the ETC tasking model, the *CLOCK* main machine and the *SCHEDULER* main machine both use hierarchical composition and need to be flattened. In the definition of the *CLOCK* main machine both rules utilize hierarchical composition in parallel, but the growth is tractable since it leads to only 16 rules (2 * 2 * 2 * 2). A similar growth occurs in the *SCHEDULER* main machine when executing the rule to wake up the various tasks. The number of rules for each flattened main machine of the TASM model is shown in Table 8.15.

The complete UPPAAL model contains 4 timed automata, including 3 automata for each main machine of the TASM model and 1 automata to enforce the "Else rule" of the *TASKS* main machine.

The UPPAAL verifier can be used to ensure that the scheduler behaves correctly according to expected functionality. For example, simple liveness queries can involve verifying that each task is eventually finished. For example, verifying that the manager task eventually executes can be expressed in the UPPAAL query language as "`A<> manager_s == 4`". Similar queries were formulated for the other tasks as well. An important safety property of the scheduler is that it enforces the single processor nature of the execution platform by not setting more than one task to the executing state. This safety assertion can be formulated as "`A[] !((monitor_s == 3 && manager_s == 3) || (monitor_s == 3 && servo_s == 3) || (manager_s == 3 && servo_s == 3))`". Other safety and liveness properties can be formulated as well, as needed.

Completeness and consistency

The completeness and consistency of the tasking model was verified successfully following the approach presented in 5.1. The results of the completeness analysis, for the machines which are not trivially complete in shown in Table 8.16. The consistency analysis results are shown in Table 8.17.

Name	Propositions	Clauses	Complete
CLOCK	3	7	Yes
SCHEDULER	4	15	Yes
SET_EXECUTION_PRIORITY	12	46	Yes

Table 8.16: Completeness analysis results for the ETC tasking model

Name	Propositions	Clauses	Consistent
CLOCK	3	7	Yes
SCHEDULER	4	15	Yes
TASKS	12	39	Yes
SET_EXECUTION_PRIORITY	12	46	Yes

Table 8.17: Consistency analysis results for the ETC tasking model

For the tasking TASM model, most machines are trivially complete because the "Else rule" is used extensively, except for the *CLOCK* main machine and for the *SET_EXECUTION_PRIORITY* sub machine. Most machines are also trivially consistent, since most machines have very few rules, except for the 3 main machines and the *SET_EXECUTION_PRIORITY* sub machine. The *TASKS* main machine is the only machine that is not consistent and the counterexample generated by the *SAT* solver is the case where two tasks are executing simultaneously. Clearly, this case is not reachable in the model since the tasking structure is executed on a single processor and this safety property of the scheduler was verified against the UPPAAL model in the previous subsection.

8.3.3 Execution Time Analysis

The execution time analysis of the tasking structure concerns whether any of the 3 tasks could ever miss a deadline given the scheduling priority and the execution time of individual tasks. For the servo task, analyzing whether there is a possibility that the task could miss a deadline can be obtained by measuring the longest running time from the task being released to the task being finished. This model path can be easily formulated using the approach described in Section 5.3.3. The resulting observer automaton to measure the time necessary to complete this path is shown in Figure 8-5.

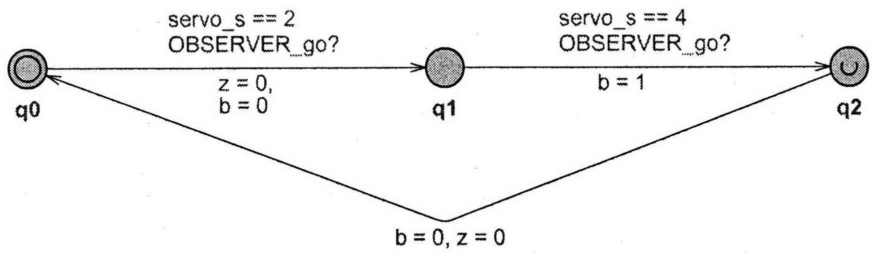


Figure 8-5: Observer automaton to verify the execution time of the servo task

After 2 iterations, the worst-case execution time of the servo task is measured to be 2100 μs, which is congruent with the preliminary analysis in the tasking model description. For the monitor task, the worst-case execution time is measured to be 1200 μs, corresponding to a one block execution delay and the worst-case execution time of the task. For the manager task, the worst-case execution time is 5 μs, with no delay before starting execution. This result is also expected since the manager task has the highest priority of all the tasks and the other tasks' execution times will fit inside the 1 ms blocks.

8.3.4 Test Case Generation

In the ETC tasking model, the TASM machines describe the behavior of a scheduler and a set of tasks with different periods. For the test case generation, it is assumed that all the machines in the TASM model represent software components. Consequently, for all the machines in the model, test suites are generated using the algorithm described in Listing 7.8. The results of the test case generation are shown in Table 8.18. In Table 8.18, the first column provides the machine name, the second column lists the number of test case templates in the test suite template for the machine, and the third column lists the number of test cases from the test suite required for unit testing of the machine. Per the approach described in Chapter 7, the test cases are generated using the rule coverage criterion, explained in Section 7.1.1.

Machine	Test Suite	Unit Testing
CLOCK	4	2
SCHEDULER	20	4
TASKS	4	4
finished_to_waiting	3	3
MANAGER_TICK	2	2
MONITOR_TICK	2	2
SERVO_TICK	2	2
SET_EXECUTING_TASK	9	2
SET_EXECUTION_PRIORITY	8	8
UPDATE_TASK_STATUSES	4	2
WAKE_UP_MANAGER	2	2
WAKE_UP_MONITOR	2	2
WAKE_UP_SERVO	2	2
WAKE_UP_TASKS	6	2

Table 8.18: Test case generation results for the ETC tasking model

A sample test case template from the *SCHEDULER* main machine test suite is shown in Table 8.19.

8.3.5 Discussion

Modeling the tasking structure and scheduler in the TASM language enabled the verification of safety and liveness of the scheduler. Furthermore, the analysis of completeness and consistency uncovered missing cases in the model during the development

Pre State	Post State	Coverage Item
scheduler_s{*wakeup*}, *oldtick*{3}, *tick*{5}, *manager_s*{*released*}, *managertick*{$MANAGER\_PERIOD$}, *monitor_s*{*released*}, *monitortick*{$MONITOR\_PERIOD$}, *servo_s*{*released*}, *servotick*{$SERVO\_PERIOD$}	*scheduler_s*{*execute*}, *oldtick*{5}, *manager_s*{*released*}, *monitor_s*{*released*}, *servo_s*{*released*}	$SCHEDULER.R_2$, $WAKE\_UP\_TASKS.R_1$, $WAKE\_UP\_MANAGER.R_1$, $WAKE\_UP\_MONITOR.R_1$, $WAKE\_UP\_SERVO.R_1$

Table 8.19: Sample test case template from the test suite for the *SCHEDULER* main machine

stages. While the scheduler is fairly straightforward since it uses a static fixed priority scheduling scheme with no preemption, modeling it in the TASM language shows the versatility of the language. Furthermore, analyzing the TASM tasking model using the framework uncovered errors in the Simulink model which were not immediately apparent. The Simulink model did not model the mutual exclusion of each task correctly. In part, this is due to the confusion surrounding the semantics of Stateflow; however, Simulink and Stateflow do not provide any verification facilities to ensure that these cases cannot happen. By modeling the scheduler and tasks in TASM and by using the proposed framework, the scheduler was verified to ensure that all tasks are executed while ensuring mutual exclusion.

The execution time analysis displays another application of the observer automaton paradigm, to evaluate whether certain deadlines are missed. The worst-case execution times for each task to complete its execution, while taking into account the priority scheme, were obtained to make sure that no deadlines are missed. Once again, this type of analysis is not possible with Simulink and Stateflow. In this case, the schedulability analysis was quite simple, and could have been done with pen and paper; however, more complex schedulability problems would not require many changes to the TASM model and the analysis approach could be reused without any modifications. The execution time analysis also uncovers some limitations in the observer automaton described in Section 5.3. Since the form of the automaton observes only simple paths between a fixed start state and a fixed final state, certain properties cannot be expressed in this paradigm. For example, it is not possible to formulate a path to measure the execution time required for all tasks to complete their execution. Formulating this property is possible through a custom observer automaton with a structure more complex than the basic structure presented in Figure 5-2. Other properties that cannot be estimated using the proposed observer automaton structure include determining the major cycle. Nevertheless, the proposed approach has yielded interesting results in verifying the absence of missed deadlines for the tasking model. Furthermore, by translating the TASM model to UPPAAL , an end-user is free to use UPPAAL freely without imposing limits to the functionality provided by the framework which is usable and flexible, but represents only a subset of the capabilities of UPPAAL .

8.4 Electronic Throttle Controller: Low Level Model

The high level version of the electronic throttle controller, presented in Section 8.2, focused on the logic used to set the mode of operation of the controller and focused on the logic used to calculate the desired current. In this version of the ETC model, the throttle controller functionality modeled in Section 8.2 is implemented using the tasking model and the scheduler modeled in Section 8.3. This model combines time-dependent behavior with the mode switching logic and with the current calculation logic. The model presented in this section also extends previous models by adding resources that are consumed by the throttle controller. The analysis presented in this section is used to illustrate the bi-directional traceability capabilities of the presented framework, as described in Chapter 6, and the generation of unit test cases, as described in Section 7.6.

8.4.1 Model

In order to model the implementation of the throttle controller using the tasking structure described in the previous section, the functional model developed in Section 8.2.1, referred to as model $\mathcal{F}_0$, is combined with the tasking model developed in Section 8.3.1, referred to as model $\mathcal{T}_0$. For the most part, the machines of model $\mathcal{F}_0$ and the machines of model $\mathcal{T}_0$ can be combined directly without changes, except for the *CONTROLLER* main machine of model $\mathcal{F}_0$ and the *TASKS* main machine of model $\mathcal{T}_0$. The datatypes and environment variables can also be combined without modification.

The resources that are included in the model are "memory" and "power". The amount of memory available for the throttle controller is 2048 kilobytes, per the properties of the target platform implementation platform [251]. For power consumption, there is typically no upper bound, so a large value is chosen as the resource upper bound, 1 Mega Watt. The case of power consumption is interesting because analyzing the power consumption is not done to establish whether a finite amount will be exhausted, but is done to understand what the peak power consumption will be. The modeling of resources is included in the "final" model resulting from the combination of model $\mathcal{F}_0$ and model $\mathcal{T}_0$, a model called model $\mathcal{FT}_1$. The "final model" is obtained through successive refinements of each model. The following subsections describe the refinements used to obtain model $\mathcal{FT}_1$.

Refined Tasking Model Level 1: $\mathcal{T}_1$

The first step of combining model $\mathcal{F}_0$ and model $\mathcal{T}_0$ is to divide the manager task of model $\mathcal{T}_0$ into two steps – one step to set the major mode and one step to set the minor mode. To maintain the semantics of the manager task, the steps should be consecutive and atomic within the manager task execution. This refinement is achieved through a "step expansion refinement", as described in Section 6.2.1. The step expansion is achieved by dividing rule R_2 of the *TASKS* main machine, shown in Listing 8.18, into 2 rules, R_{11} and R_{12}. The consecutive execution of the two refined rules is achieved by introducing a new variable called **manager_s_step**, whose responsibility is to ensure that rule R_{11} and rule R_{12} are executed in sequence. The modified rules of the refined model, called model $\mathcal{T}_1$, are shown in Listing 8.19.

Listing 8.19 Rules of the *TASKS* main machine of model $\mathcal{T}_1$ (partial)

```
R11: Execute manager - set major mode
{
  t := [0, 3];

  if manager_s = executing and manager_s_step = major_mode then
    manager_s_step := minor_mode;
}

R12: Execute manager - set minor mode
{
  t := [0, 2];

  if manager_s = executing and manager_s_step = minor_mode then
    manager_s_step  := major_mode;
    manager_s       := finished;
}
```

The traceability relationship from model $\mathcal{T}_0$ to model $\mathcal{T}_1$ is simply the identity mapping of all rules of all machines except for rule R_1 of the *TASKS* main machine, whose traceability mapping is included in the set of "step expansion refinements". The complete traceability relationship can be expressed as:

$$\mathrm{T} = \mathcal{T}_0 \leftrightarrow \mathcal{T}_1 = \mathbb{T}_{id} \cup \mathbb{T}_{sexp}$$

Where:

- $\mathbb{T}_{id}$ = identity mappings for all rules of all machines except for rule R_1 of machine *TASKS*

- $\mathbb{T}_{sexp} = (\langle \{TASKS.R_1\}, \{TASKS.R_{11}, TASKS.R_{12}\} \rangle)$

It is fairly obvious to see that the correctness criteria of Section 6.2.2 for the step expansion refinement is met since the time annotations of rule R_{11} and rule R_{12} are contained within the time annotation of rule R_1. Furthermore, the guards of the refined rules also meet the correctness criteria, as shown by the truth table shown in Table 8.20.

manager_s = executing	manager_s_step = major_mode	manager_s_step = minor_mode	G_1	G_{11}	G_{12}	$(G_{11} \vee G_{12})$
T	T	F	**T**	T	F	**T**
T	F	T	**T**	F	T	**T**
F	T	F	**F**	F	F	**F**
F	F	T	**F**	F	F	**F**

Table 8.20: Truth table for the step expansion refinement between rule R_1 of model $\mathcal{T}_0$ and rules R_{11} and R_{12} of model $\mathcal{T}_1$

Furthermore, no other rule changes the value of `manager_s` from `executing` to another value, no other rule changes the value of `manager_s_step`, the two rules are sequential through `manager_s_step`, and the result of executing both rules in sequence is the same as the result of executing rule R_1. Because the correctness criteria holds for the step expansion refinement, the analysis performed on model $\mathcal{T}_0$, in Section 8.3, still holds in model $\mathcal{T}_1$. Since the time annotations in both models are equivalent through equality bounds, the analysis results that are preserved from model $\mathcal{T}_0$ include the functional analysis, the schedulability analysis, and the worst-case execution time analysis. Consequently, the absence of deadlines and the safety of the scheduler, as determined in Section 8.4.4, are preserved through the refinement, in model $\mathcal{T}_1$.

Refined Tasking Model Level 2: $\mathcal{T}_2$

In the second refinement level, elements manipulating the state from model $\mathcal{F}_0$ are introduced into model $\mathcal{T}_1$. The first refinement for this level happens with the *SCHEDULER* main machine. The refinement is of the type "step expansion", performed by adding rule R_0 to the rules of the *SCHEDULER* main machine to be executed after rule R_4 and before rule R_1. The refinement is achieved by expanding the `Scheduler` datatype by adding an extra member, called **update_state**. The refinement occurs so that the scheduler reads the state of the sensors at 1 ms intervals, at the beginning of each tick. This is achieved by reusing the *SAMPLE_STATE* sub machine defined in Section 8.2. The added rule, rule R_0, and the modified version of rule R_4 are shown in Listing 8.20. The refinement adds a step to the scheduler execution, at the beginning, by caching the state at the beginning of each execution cycle. The state updating rate is congruent to the functionality of the Simulink model described in [111], which is driven by a 1 ms clock.

Listing 8.20 Modified rules of *SCHEDULER* main machine of model $\mathcal{T}_1$

```
R0: Step 0, update state
{
  if scheduler_s = update_state then
    SAMPLE_STATE();
    scheduler_s := update_tasks;
}

...

R4: Wait for a tick
{
  t := next;

  if scheduler = wait then
    scheduler_s := update_state;
}
```

Using a structure and an argument similar to the step expansion refinement for the *TASKS* main machine in model $\mathcal{T}_1$, it can be shown that the step expansion refinement preserves the semantics of model $\mathcal{T}_1$ and, by associativity, the semantics of model $\mathcal{T}_0$. The refinement meets the correctness criteria of Section 6.2.2, since the refined rules follow sequentially, and the updates introduced by the *SAMPLE_STATE* sub machine affect only the expanded state space and no other rule in model $\mathcal{T}_1$.

The second refinement that occurs at this level is the addition of state manipulating functionality for the task executions. Analogously to the refinement of the *SCHEDULER* main machine, this refinement is achieved by reusing sub machines from model $\mathcal{F}_0$, namely the *SET_MAJOR_MODE*, *SET_MINOR_MODE*, *MONITOR_HEALTH*, and *CALCULATE_OUTPUT* sub machines. The refined *TASKS* main machine is shown in Listing 8.21. For this refinement, the refinement is of type "rule expansion", where each rule R_i of the *TASKS* main machine of model $\mathcal{T}_1$ is mapped one-to-one to each rule of the TASKS main machine of model $\mathcal{T}_2$. It is easy to see that, in the mapping, each time annotation and each guard is equivalent. The difference happens in the effect expressions. Since the sub machines that are used in the refinement do not depend or modify the state of model $\mathcal{T}_1$, it can be shown that the refinement preserves the semantics of model $\mathcal{T}_1$, and by associativity, the semantics of model $\mathcal{T}_0$. The equivalence of the guards in the mapping can be easily visualized through previous analysis. Per the analysis presented in Table 8.9, all the sub machines introduced in Listing 8.21 are complete. Consequently, the equivalent "flattened" machine for Listing 8.21 will yield a machine where the guards between the mapped rules will also be complete since completeness is preserved through hierarchical composition per the proof of Theorem 5.1. The extra rules introduced by the use of sub machines are included in the rule expansion refinement since the equivalent "flattened" machine would yield the one-to-many mapping that includes the extra rules.

The traceability relationship for this refinement level can be expressed as:

$$\mathrm{T} = \mathcal{T}_1 \leftrightarrow \mathcal{T}_2 = \mathbb{T}_{id} \cup \mathbb{T}_{sexp} \cup \mathbb{T}_{rexp}$$

Where:

- $\mathbb{T}_{id}$ = identity mappings for all rules of all machines except for rule R_4 of machine *SCHEDULER*, and rules R_{11}, R_{12}, R_2, and R_3 of machine *TASKS*

- $\mathbb{T}_{sexp} = (\langle\ \{SCHEDULER.R_4\},\ \{SCHEDULER.R_4,\ SCHEDULER.R_0\}\ \rangle)$

- $\mathbb{T}_{rexp} = (\langle\{TASKS.R_{11}\},\ \{TASKS.R_{11}\}\ \rangle,\ \langle\{TASKS.R_{12}\},\ \{TASKS.R_{12}\}\ \rangle,$

Listing 8.21 Modified *TASKS* main machine of model $\mathcal{T}_2$

```
R11: Execute manager
{
  t := [0, 3];

  if manager_s = executing and manager_s_step = major_mode then
    SET_MAJOR_MODE();
    manager_s_step := minor_mode;
}

R12: Execute manager
{
  t := [0, 2];

  if manager_s = executing and manager_s_step = minor_mode then
    SET_MINOR_MODE();
    manager_s_step  := major_mode;
    manager_s       := finished;
}

R2: Execute monitor
{
  t := [100, 200];

  if monitor_s = executing then
    MONITOR_HEALTH();
    monitor_s := finished;
}

R3: Execute servo
{
  t := [70, 100];

  if servo_s = executing then
    CALCULATE_OUTPUT();
    servo_s := finished;
}

R4: Else, do nothing, wait for event
{
  t := next;

  else then
    skip;
}
```

$\langle\{TASKS.R_2\}, \{TASKS.R_2\}\rangle, \langle\{TASKS.R_3\}, \{TASKS.R_3\}\rangle\rangle$

Through this sequence of refinements of model $\mathcal{T}_0$, the refinements have conformed to the correctness criteria described in Section 6.2.2. Consequently, the analysis results performed on model $\mathcal{T}_0$ are preserved for model $\mathcal{T}_2$.

Refined Functional Model Level 1: $\mathcal{F}_1$

The refinement of model $\mathcal{T}_0$ from a tasking model to a combination of tasking and functional model was performed in appropriately defined steps to show that the refinements were performed to preserve the analysis performed on model $\mathcal{T}_0$. In this section, an attempt is made to refine model $\mathcal{F}_0$ stepwise to preserve its semantics all the while introducing scheduling. The first step of the refinement introduces time annotations into the *CONTROLLER* main machine using the task execution times. The modified rules of the *CONTROLLER* machine are shown in Listing 8.22. In Listing 8.22, the only difference in the refinement is the addition of time annotations for rules R_2, R_3, R_4, and R_5, using the task execution times. This refinement corresponds to a rule expansion refinement between the *CONTROLLER* main machine of model $\mathcal{F}_0$ and the *CONTROLLER* main machine of model $\mathcal{F}_1$.

Because the guards and effect expressions are equivalent between model $\mathcal{F}_0$ and model $\mathcal{F}_1$, the correctness criteria of the rule expansion refinement for the guards and effect expressions are met. However, the correctness criteria regarding the time annotations between model $\mathcal{F}_0$ and model$\mathcal{F}_1$ are not met. Consequently, if timing analysis was performed in model $\mathcal{F}_0$, the analysis results would not necessarily hold in $\mathcal{F}_1$. However, while the addition of time annotations affects the timing of the model, the sequence of execution of the rules remains unchanged between model $\mathcal{F}_0$ and model $\mathcal{F}_1$. Consequently, the functional analysis performed on model $\mathcal{F}_0$, in Section 8.2.2 still holds in model $\mathcal{F}_1$. This can be established using the argument that the *CONTROLLER* main machine executes a "full loop" using the cached state. In the case of model $\mathcal{F}_1$, the performance of the controller will be lower since the

Listing 8.22 Rules of the *CONTROLLER* main machine of model $\mathcal{F}_1$

```
R1: Controller loop when nominal
{
  if control_mode = sample then
    SAMPLE_STATE();
    control_mode := mode_set;
}

R2: Controller loop to set major mode
{
  t := [0, 3];

  if control_mode = mode_set_major then
    SET_MAJOR_MODE();
    control_mode := mode_set_minor;
}

R3: Controller loop to set minor mode
{
  t := [0, 2];

  if control_mode = mode_set_minor then
    SET_MINOR_MODE();
    control_mode := output;
}

R4: Controller loop to output current
{
  t := [70, 100]

  if control_mode = output then
    CALCULATE_OUTPUT();
    control_mode := health;
}

R5: Controller loop to find failure
{
  t := [100, 200];

  if control_mode = health then
    MONITOR_HEALTH();
    control_mode := sample;
}
```

actions are not instantaneous, but the safety and liveness of the ETC are preserved because the correctness criteria hold. The traceability relationship between model $\mathcal{F}_0$ and model $\mathcal{F}_1$ can be expressed as:

$$\mathrm{T} = \mathcal{F}_0 \leftrightarrow \mathcal{F}_1 = \mathbb{T}_{id} \cup \mathbb{T}_{rexp}$$

Where:

- $\mathbb{T}_{id}$ = identity mappings for all rules of all machines except for rules R_1, R_2, R_3, R_4, and R_5 of machine *CONTROLLER*
- $\mathbb{T}_{rexp} = (\langle\{CONTROLLER.R_1\}, \{CONTROLLER.R_1\}\rangle,$
  $\langle\{CONTROLLER.R_2\}, \{CONTROLLER.R_2\}\rangle,$
  $\langle\{CONTROLLER.R_3\}, \{CONTROLLER.R_3\}\rangle,$
  $\langle\{CONTROLLER.R_4\}, \{CONTROLLER.R_4\}\rangle,$
  $\langle\{CONTROLLER.R_5\}, \{CONTROLLER.R_5\}\rangle)$

Refined Functional Model Level 2: $\mathcal{F}_2$

The refinement of model $\mathcal{F}_1$ into model $\mathcal{F}_2$ involves adding the scheduler and the tasking structure of model $\mathcal{T}_0$ to model $\mathcal{F}_1$. The refined version of the *CONTROLLER* main machine resembles the machine of Listing 8.21, with the variable `control_mode` removed from the rules. Since model $\mathcal{F}_2$ is drastically different from model $\mathcal{F}_1$, the functional properties analyzed in model $\mathcal{F}_0$ no longer hold in model $\mathcal{F}_2$. Nevertheless, the rules can be traced syntactically using rule expansion refinements with an identical mapping from model $\mathcal{F}_1$ to model $\mathcal{F}_2$ as in the mapping between model $\mathcal{F}_0$ and model $\mathcal{F}_1$.

Final Refined Model: $\mathcal{FT}_1$

The two refinement branches converge at a common model called $\mathcal{FT}_0$ that is traceable to model $\mathcal{F}_2$ and, by associativity, to model $\mathcal{F}_0$. Furthermore, model $\mathcal{FT}_0$ to model $\mathcal{T}_2$. The final step in refining the ETC low level model, resulting in model $\mathcal{TF}_1$, adds resource annotations to model $\mathcal{FT}_0$. The model contains 2 resources – memory and power consumption. The resources are modeled to estimate the maximum power consumption of the throttle controller, and to ensure that the memory used by the controller is adequately bounded. The power consumption resource utilization was estimated using the characteristics of the Xilinx Virtex II Pro implementation platform, which uses a PowerPC 405 processor operating at 10 MHz [251]. The memory consumption was also estimated using the properties of the Xilinx Virtex II Pro, using a combination of the operations listed in Table 8.14 and the Simulink model. Both resources are consumed only by the functional aspects of the model, that is, mode setting, desired current calculation, and health monitoring. The resource consumption of the scheduler and the clock are assumed to be negligible.

In order to estimate the memory necessary to set the mode and to calculate the desired current, the Simulink model is abstracted using the list of Simulink blocks, variables, and parallel operations performed in each subsystem. For example, the sliding mode controller used to calculate the throttle controller current commanded by the driver input is shown in Figure 8-6. In Figure 8-6, there are 6 parallel lines of computation and the computations are achieved through 26 variables of type float, 3 integrators, 9 gains, 7 additions, 1 saturation, 2 sign reversals, 1 discrete filter, and a summation at the end of the computation. The memory usage is estimated using the number of variables necessary to perform the calculation and the properties of the execution platform. The power consumption is estimated using the parallel branches of computation, the number of maximum number of clock cycles required to perform on each branch, and the properties of the implementation platform. For most current calculations in the Simulink model, there are multiple calculation paths that can be followed to obtain the desired current. Consequently, it is possible to obtain

Calculated Current	Min Power (mW)	Max Power (mW)	Min Memory (bytes)	Max Memory (bytes)
Human	769	895	196	360
Cruise	800	800	128	128
Human + Cruise	864	1695	324	826
Traction	800	800	128	128
Rev	800	800	128	128
Traction + Rev	1425	1425	648	648
Fault	855	895	512	512

Table 8.21: Resource usage estimates for the ETC low level model

minimum and maximum values for both resource calculations. The results of the resource usage estimation for the desired current calculations are shown in Table 8.21. Similar estimates were made for the health monitoring logic and for the startup and shutdown modes. The results of the estimates serve as resource annotations in the TASM model. The modified machines containing resource annotations are provided in Section E.3.

Complete Model

The complete ETC low level TASM model contains 5 main machines, 14 function machines, and 20 sub machines. The complete low level ETC model is documented in Appendix E where the list of all machines is shown in Table E.3 and in Table E.4.

8.4.2 Traceability

The sequence of refinements performed in Section 8.4.1 yields a tree of models that can be navigated through the relation of rules of machines. The complete traceability relationship can be visualized as illustrated in Figure 8-7. In Figure 8-7, model $\mathcal{F}_0$ corresponds to the model described and analyzed in Section 8.2, model $\sqcup_0$ corresponds to the model described and analyzed in Section 8.3, and model $\mathcal{FT}_1$ corresponds to the model described and analyzed in this section.

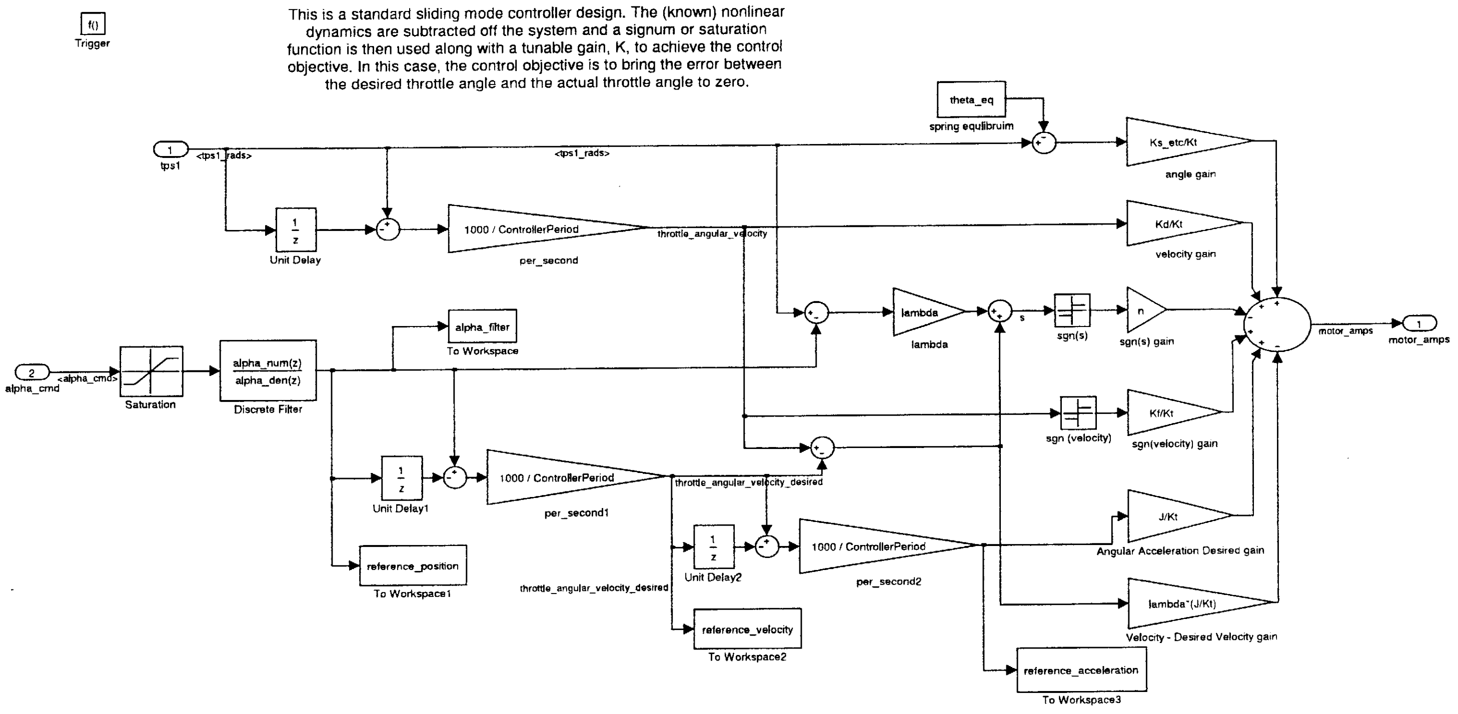

This is a standard sliding mode controller design. The (known) nonlinear dynamics are subtracted off the system and a signum or saturation function is then used along with a tunable gain, K, to achieve the control objective. In this case, the control objective is to bring the error between the desired throttle angle and the actual throttle angle to zero.

Figure 8-6: Simulink sliding mode controller to calculate driver throttle current

$$\begin{tikzcd}
T_0 \arrow[d, leftrightarrow] & & F_0 \arrow[d, leftrightarrow] \\
T_1 \arrow[d, leftrightarrow] & & F_1 \arrow[d, leftrightarrow] \\
T_2 \arrow[dr, leftrightarrow] & & F_2 \arrow[dl, leftrightarrow] \\
& TF_0 \arrow[d, leftrightarrow] & \\
& TF_1 &
\end{tikzcd}$$

Figure 8-7: Traceability between different versions of the ETC model

8.4.3 Functional Analysis

Since the steps of the refinements performed to combine the functional model presented in Section 8.2 with the model presented in Section 8.3 do not preserve the semantics of model $\mathcal{F}_0$, the functional analysis performed in Section 8.2.2 must be performed again on model $\mathcal{TF}_1$. The safety properties verified in $\mathcal{F}_0$ utilized the sequential nature of the controller execution to ensure that the mode was always set correctly according to the requirements, based on the state cached by the controller and based on the step of operation of the controller. However, in model $\mathcal{TF}_1$, since the functionality is implemented using a tasking model where the manager task executes at a slower rate (every 10 ms) than the controller resolution (1 ms), some controller iterations do not meet the safety properties expressed in Section 8.2.2. The properties verified in Section 8.2.2 assumed that the controller would set the mode at every controller iteration, which is not true for model $\mathcal{TF}_1$. Consequently, the queries from Section 8.2.2 need to be modified. The queries can capitalize on the subtleties of the execution, where the state is cached at the beginning of the controller iteration and each task will be in the *finished* state in an iteration only if the task has executed during the previous iteration. Furthermore, since it has been established through the execution time analysis of model $\mathcal{T}_0$ that the execution of each task will complete within a controller iteration, the safety property to verify that the limiting mode is always set correctly, when the engine speed or the torque is above the predefined threshold, can be formulated as:

- $A\ G\ (system\_health = nominal \wedge c\_engine\_speed > MAX\_ENGINE\_SPEED \wedge scheduler\_s = update\_state \wedge manager\_s = finished) \rightarrow (controller\_mode = limiting)$

- $A\ G\ (system\_health = nominal \wedge c\_vehicle\_torque > MAX\_TORQUE \wedge scheduler\_s = update\_state \wedge manager\_s = finished) \rightarrow (controller\_mode = limiting)$

The scheduler will be in the "update state" step *after* an iteration has completed and *before* the cached state is updated. Consequently, at that step, if the manager task has executed, the mode should be set correctly. The requirement that no fault be present is necessary because if a fault is detected, it will override all other modes. Similar queries can be formulated to verify that the mode is set correctly for the driving mode and for the minor modes. To verify the calculation of the desired current, the queries from Section 8.2.2 can be adapted in a similar fashion:

- $A\ G\ (controller\_mode = driving \land scheduler\_s = update\_state \land servo\_s = finished) \rightarrow (desired\_current = human\_c \lor desired\_current = cruise\_c \lor desired\_current = max\_driving\_c)$

- $A\ G\ (controller\_mode = limiting \land cruise\_mode = active \land rev\_limiting\_mode = active \land traction\_mode = active \land scheduler\_s = update\_state \land servo\_s = finished) \rightarrow (desired\_current = min\_limiting\_c)$

The status of the servo task also needs to be queried because the servo task will not execute at every controller iteration and will execute in an iteration only if it is the only released task since it has the lowest priority of all the tasks. Other similar queries could be written to verify that the minor modes are set properly and that the current is calculated correctly for all the combination of major and minor modes. Those queries would follow a pattern similar to the queries presented in this section and are omitted for brevity. In order to verify the safety properties, the TASM model is translated to UPPAAL 's timed automata.

UPPAAL Model

In order to model check the TASM model for safety assertions, the ETC TASM model leverages the UPPAAL model checker, using the translation approach documented in Appendix C. Because the timed automata used in UPPAAL do not have hierarchical

Machine	Rules	Flattened Rules
CLOCK	2	16
DRIVER	11	11
SCHEDULER	5	29
TASKS	5	33
VEHICLE	9	9

Table 8.22: Number of rules for flattened main machines

composition facilities, the TASM main machines need to be "flattened" per the approach in the proof of Theorem 4.1 and in the proof of Theorem 4.2. In the low level ETC model, model $\mathcal{TF}_1$, 3 main machines use hierarchical composition and need to be flattened – the *CLOCK* main machine, the *SCHEDULER* main machine, and the *TASKS* main machine. The number of rules for each flattened main machine of the TASM model is shown in Table 8.22.

The complete UPPAAL model contains 9 timed automata, including 5 automata for each main machine of the TASM model and 4 automata to enforce the "Else rules" of the *DRIVER*, *SCHEDULER*, *TASKS*, and *VEHICLE* main machines. The safety properties given as temporal logic formulas can be easily translated to the TCTL query language of UPPAAL . The UPPAAL queries corresponding to the safety properties stated above are shown below, in the order in which they were introduced:

- `A[] (c_engine_speed > MAX_ENGINE_SPEED && system_health == 1 && scheduler_s == 5 && manager_s == 4) imply (controller_mode == 5)`

- `A[] (c_vehicle_torque > MAX_TORQUE && system_health == 1 && scheduler_s == 5 && manager_s == 4) imply (controller_mode == 5)`

- `A[] (controller_mode == 4 && scheduler_s == 5 && servo_s == 4) imply (desired_current == 2 || desired_current == 3 ||`

```
desired_current == 7)
```

- `A[] (controller_mode == 5 && cruise_mode == 1 && rev_limiting_mode == 1 && traction_mode == 1 && scheduler_s == 5 && servo_s == 4) imply (desired_current == 6)`

These properties were successfully verified by running the queries through the UPPAAL verifier. The model can also be queried to verify certain liveness properties, to ensure that the model behaves correctly. For example, the property "eventually, the car can be in cruise control" can be verified. This property can be formulated as the liveness property "`E<> cruise_mode == 1`". Other liveness and safety properties can be formulated in a similar fashion, as needed.

Completeness and Consistency

The completeness and consistency of the low level ETC model was verified successfully following the approach presented in 5.1. The results of the completeness analysis are shown in Table 8.23. The consistency analysis results are shown in Table 8.24. The completeness and consistency analysis was performed only for the machines whose rules were changed from the model presented in Section 8.2 and the model presented in Section 8.3. For the machines in the low level model, model $\mathcal{FT}_1$, whose rules were not modified, the completeness and consistency results are available in Table 8.9, Table 8.16, Table 8.10, and Table 8.17.

Name	Propositions	Clauses	Complete
SCHEDULER	5	32	Yes
TASKS	N/A	N/A	Yes

Table 8.23: Completeness analysis results for the ETC low level model

For the low level ETC TASM model, the only machines whose completeness and consistency need to be verified are the *SCHEDULER* main machine and the *TASKS*

Name	Propositions	Clauses	Consistent
SCHEDULER	5	32	Yes
TASKS	14	42	No

Table 8.24: Consistency analysis results for the ETC low level model

main machine. The other machines which were modified were altered to include resource consumption annotations to the rules. Because the rule guards were not changed, the completeness and consistency of the machines are not affected. According to the results presented in Table 8.24, the *TASKS* main machine is not consistent. The counterexample generated by the *SAT* solver is the same state as the one described in the analysis of the consistency results presented in Table 8.10. The counterexample is the state where two tasks are executing simultaneously, a state which is clearly not reachable since the functionality is implemented on a single processor architecture. It can be easily verified that this state is not reachable, through the safety invariant query "`A[] !((monitor_s == 3 && manager_s == 3) || (monitor_s == 3 && servo_s == 3) || (manager_s == 3 && servo_s == 3))`" against the timed automata UPPAAL model. Interestingly enough, this safety invariant was verified for model $\mathcal{T}_0$ in Section 8.3.1 and given the nature of the refinements, this property is guaranteed to hold in model $\mathcal{TF}_1$.

8.4.4 Execution Time Analysis

Given that the refinements of the tasking model, model $\mathcal{T}_0$, presented in Section 8.3, meet the correctness criteria, the execution time analysis results from Section 8.3.3 are preserved in model $\mathcal{FT}_1$. Consequently, the schedulability analysis and the absence of missed deadlines analysis performed on model $\mathcal{T}_0$ are guaranteed to hold in model $\mathcal{TF}_1$. Using the approach presented in Section 5.3, other execution time properties of model $\mathcal{TF}_1$ can be verified. For example, execution time properties for a combination of functional properties and scheduling properties can be verified.

End-to-End Latency

End-to-end latency refers to the amount of time that it takes for the system to react to a change in the environment. In terms of the ETC, end-to-end latency could refer to the amount of time required for the controller to output the appropriate current to remedy a given situation. For example, the end-to-end latency of mitigating the torque rising over the critical threshold would be the BCET and WCET of the ETC setting the desired current to "traction limiting", taking into account the effects of scheduling to set the appropriate controller mode and to calculate the desired current based on the mode. The *DRIVER* and *VEHICLE* main machines model the behavior of the surroundings of the ETC and are modeled so that changes in the vehicle torque and in the engine RPM can occur at any point. Consequently using the iterative bounded liveness approach against the UPPAAL model derived in Section 8.4.3 can yield the desired measures through an appropriately defined observer automaton. For example, the observer automaton shown in Figure 8-8 is used to measure the end-to-end latency of the ETC outputting current to mitigate a vehicle torque over the critical threshold. In the observer, it is important to include the health of the system in the path formula. Otherwise, the timing of the path could be unbounded because a fault could occur after the torque rise and the ETC would enter the "faulty" mode and would never handle the torque condition.

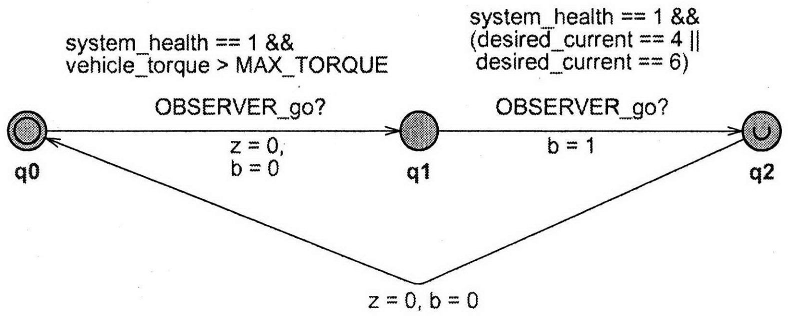


Figure 8-8: Observer automaton to measure the end-to-end latency of the ETC for the vehicle torque being over the critical threshold

After two iterations of verifying ϕ_{max} against the UPPAAL model, the WCET of the end-to-end latency corresponding to the behavior observed through the automaton shown in Figure 8-8 is determined to be 12100μs. This WCET corresponds to the rise in torque occurring at the same time as the manager task beginning execution, after the state has been cached. 10000μs elapse until the next execution of the manager task (10 ms period), where the appropriate controller mode is set after the manager task has completed its execution. However, the desired current will not be calculated until the servo task completes its execution. Since the period of the servo task is (3 ms), the servo task will be released 2000μs after the beginning of the execution of the manager task. Since the servo task has a WCET of 100μs, the desired current to handle the torque limiting will occur 100μs after the beginning of the execution of the servo task, resulting in an end-to-end latency WCET of 12100μs. It is interesting to notice that the WCET occurs only if the torque rise occurs at the beginning of the first execution of the manager task within the major cycle. Since the major cycle is 30 ms, the manager task executes 3 times per major cycle, at the beginning, 10 ms into the cycle, and 20 ms into the cycle. The WCET occurs this way because the second execution provides the largest gap between the end of the manager task and the beginning of the servo task, when the mode is set and the desired current is calculated.

The BCET of the same end-to-end latency is verified to be 2070μs, also after two iterations, but through verifying ϕ_{min}. This BCET corresponds to the rise in torque occurring immediately before the beginning of the major cycle, when all tasks are released. Given the task priority, the manager task executes first, followed by the monitor task, 1000μs later, followed by the servo task, 1000μs later. Given that the BCET of the servo task is 70μs, the total BCET reaches 2070μs.

For the situation where the ETC remedies a rise in engine RPM, the BCET and WCET of the end-to-end latency is exactly the same as for the torque case. The similarity occurs because the reaction delay is purely a result of the scheduling properties of the various tasks. The time required to set the controller mode and the time required to calculate the desired current is independent of what mode is being set and which desired current is being output.

Another interesting situation to consider, and which yields different timing results, is the BCET and WCET of the end-to-end latency for the ETC to remedy a fault. The WCET for this situation is 42100μs and the BCET is 12070μs. The BCET is achieved when the fault occurs right before the beginning of the major cycle. However, since the manager task has a higher priority than the monitor task, it runs first, without the fault having been detected. The monitor task runs second and detects the fault. However, the manager task does not set the appropriate fault handling mode until 10 ms after the beginning of the major cycle. Furthermore, the appropriate desired current doest not get output until the next execution of the servo task, 2 ms after the beginning of execution of the manager task. The WCET occurs when the fault happens right after the beginning of the major cycle, once the state has been cached. The fault will not be detected until the beginning of the following major cycle, 30 ms later, and, given the priority of the manager task and the same argument as in the BCET case, the fault is not handled until 12100μs later, resulting in the WCET of 42100μs. The summary results for the end-to-end latency analysis of model $\mathcal{FT}_1$ are summarized in Table 8.25.

p_0	p_1	WCET	Iter	BCET	Iter
system_health = nominal, vehicle_torque > MAX_TORQUE	(desired_current = traction_c or desired_current = min_limiting_c), system_health = nominal	12100μs	2	2070μs	2
system_health = nominal, engine_speed > MAX_ENGINE_SPEED	(desired_current = rev_c or desired_current = min_limiting_c), system_health = nominal	12100μs	2	2070μs	2
fault = True	desired_current = fault_c	42100μs	2	12070μs	2

Table 8.25: End-to-end latency analysis results for the ETC low level model

8.4.5 Resource Consumption Analysis

The resource consumption analysis provided by the proposed framework provides calculations for the best-case and worst-case amounts of resources consumed, as explained in Section 5.4. The algorithm iterates through combinations of rules that can be executed in parallel in order to find a combination of rules who are satisfiable and which yield maximal and minimal resource usage. Per TASM semantics, resource usage is additive through parallel rule execution. The parallelism can be achieved

Resource	Type	Amount	State
power	min	769	servo_s = executing, controller_mode = driving, cruise_mode = inactive, c_pedal_angle != 0
power	max	1695	servo_s = executing, controller_mode = driving, cruise_mode = active, c_pedal_angle != 0
memory	min	128	servo_s = executing, c_vehicle_speed = 0, c_break_pedal = active, c_cruise_switch = off
memory	max	1024	monitor_s = executing, fault = True

Table 8.26: Resource usage analysis results for the ETC low level model

either through main machines executing simultaneously, or through multiple units of hierarchical composition. In the ETC model, the implementation of the functionality is fairly simple, on a single processor architecture, and there are few interactions between the tasks. Furthermore, the calculation logic for the desired current is also of low complexity.

Because the ETC implementation does not contain any parallel consumption of resources through the use of multiple main machines, the minimum and maximum amounts of resources consumed will correspond to the minimum and maximum amounts contained in an individual rule. For the maximum amount of power consumed, 1695 milliWatts, it occurs when rule R_1 of the *Driving_Throttle_C* function machine is executed. The minimum amount of power consumed is trivially 0 milliWatts, before the ETC initializes or after shutdown. The minimum non-zero power consumption is 769 milliWatts, which occurs when rule R_1 of the *Driver_Throttle_C* function machine is executed. Similar reasoning can be applied to the memory consumption. The results of the analysis, alongside the complete state of the "flattened" machine yielding the minimum and maximum values are shown in Table 8.26.

8.4.6 Test Case Generation

Since the ETC low level model, model $\mathcal{FT}_1$, combines features of the high level model, model $\mathcal{F}_0$, and features of the tasking model, $\mathcal{T}_0$, through a series of refinements, the test cases generated previously can be reused. More specifically, test suites have already been generated for these two models, in Section 8.2.3 and in Section 8.3.4. In this section, test cases are generated for the combined model using the set of refinements and the regression test case strategy described in Section 7.6. The results of the test case generation are shown in Table 8.27 and in Table 8.28. In the tables, the first column provides the machine name, the second column lists the number of test case templates in the test suite for the machine, and the third column lists the number of test cases from the test suite required for unit testing of the machine. The fourth column denotes how many test cases were regenerated and the fifth column describes the number of test cases that need to be executed, under the assumption that the test suites generated in Section 8.2.3 and in Section 8.3.4 were already executed on the respective models. Per the approach described in Chapter 7, the test cases are generated using the rule coverage criterion, explained in Section 7.1.1.

Machine	Test Suite	Unit Testing	New	Execute
CLOCK	4	2	0	0
SCHEDULER	21	5	1	2
TASKS	33	5	32	4 (32)
Cruise	2	2	0	0
Cruise_Mode	3	3	0	0
Cruise_Throttle_C	1	1	0	0
Driver_Throttle_C	1	1	0	0
Driving_Throttle_C	3	3	0	0
Fault	2	2	0	0
Limiting_Throttle_C	6	4	0	0
Over_Rev	2	2	0	0
Over_Rev_Mode	3	2	0	0
Over_Rev_Throttle_C	1	1	0	0
Over_Torque	2	2	0	0

Table 8.27: Test case generation results for the ETC low level model (part 1)

For the *SCHEDULER* main machine, a test case template is added to the test suite, using the approach for the step expansion refinement. The new test case tem-

Machine	Test Suite	Unit Testing	New	Execute
Over_Torque_Mode	3	2	0	0
Over_Torque_Throttle_C	1	1	0	0
finished_to_waiting	3	3	0	0
CALCULATE_OUTPUT	15	6	0	0
DO_SHUTDOWN	2	2	0	0
DO_STARTUP	2	2	0	0
HANDLE_FAULT	2	2	0	0
MANAGER_TICK	2	2	0	0
MONITOR_HEALTH	3	2	0	0
MONITOR_TICK	2	2	0	0
SAMPLE_STATE	1	1	0	0
SERVO_TICK	2	2	0	0
SET_EXECUTING_TASK	9	2	0	0
SET_EXECUTION_PRIORITY	8	8	0	0
SET_MAJOR_MODE	11	3	0	0
SET_MAJOR_MODE_WORK	9	7	0	0
SET_MINOR_MODE	3	2	0	0
SET_MINOR_MODE_WORK	3	1	0	0
UPDATE_TASK_STATUSES	4	2	0	0
WAKE_UP_MANAGER	2	2	0	0
WAKE_UP_MONITOR	2	2	0	0
WAKE_UP_SERVO	2	2	0	0
WAKE_UP_TASKS	6	2	0	0

Table 8.28: Test case generation results for the ETC low level model (part 2)

plate which is added covers the new rule to cache the step at the beginning of the execution of the scheduler, rule R_0. Furthermore the test case which covers the last rule in the execution sequence, rule R_4, needs to be regenerated and executed to validate the change. For *TASKS* main machine, the generation of test cases is slightly more complex because it involves integrating functionality from model $\mathcal{F}_0$ into model $\mathcal{T}_0$. Consequently, the integration test generation algorithm can be leveraged by combining the test suites of the sub machines from model $\mathcal{F}_0$ with the coverage items of the *TASKS* main machine. In Table 8.27, the test cases that need to be executed are 4 test cases to cover the changes in the *TASKS* machine and 32 test cases to cover all of the rules of the *TASKS* machine and all other sub machines introduced during the refinement. For example, a sample test case template from the *TASKS* main machine test suite is shown in Table 8.29. This test case covers rule R_{11} of machine *TASKS* and 2 rules of some of the sub machines introduced in the refinement.

Pre State	Post State	Coverage Item
$manager\_s\{executing\}$, $manager\_s\_step\{major\_mode\}$, $system\_health\{nominal\}$, $controller\_mode\{startup\}$, $startup\_done\{True\}$	$manager\_s\_step\{minor\_mode\}$, $controller\_mode\{driving\}$	$TASKS.R_{11}$, $SET\_MAJOR\text{-}\_MODE.R_1$, $SET\_MAJOR\text{-}\_MODE\_WORK.R_2$

Table 8.29: Sample test case template from the test suite for the *TASKS* main machine

8.4.7 Discussion

The Electronic Throttle Controller (ETC) proved to be an interesting cases study because it combined different facets of real-time systems such as scheduling aspects and functional aspects. Furthermore, the ETC lent itself nicely to the traceability approach described in Chapter 6. The low level model of the ETC provided insightful analysis results, especially with regards to end-to-end latency. As was uncovered in Section 8.4.4, the latency for fault detection is unnecessarily high in both the worst case and the best case. The reason for this situation is because the monitor task has a lower priority than the manager task. It would be quite simple to modify the TASM model and associated UPPAAL model such that the monitor task has higher priority than the manager task. With this change, the impact on end-to-end latency could be immediately analyzed. Measuring end-to-end latency is not possible with Simulink and it would need to be obtained through an appropriate simulation scenario or through separate analysis.

While the presented framework, the TASM language, and the associated toolset do not have the rich library of mathematical components from control theory provided by Simulink, the analysis capabilities of the framework proved to be a useful complement to the Simulink model. Matlab and Simulink provide robust facilities for control-theoric analysis of control systems, through simulation scenarios. However, the ETC analysis, as performed on the high level model, tasking model, and low level model, is not possible with Matlab and Simulink. Matlab also does not provide traceability or refinement capabilities.

The ETC low level model was derived by combining the high level model of Section 8.2 with the tasking model of Section 8.3, and through the addition of resource consumption. The traceability approach from the tasking model to the functional model proved useful as properties from the tasking model, such as schedulability analysis and execution time analysis, were preserved through the refinements. This situation occurred because the various refinements from the tasking model complied with the correctness criteria described in Section 6.2.2. While the set of correctness criteria seemed a bit restrictive when they were defined, they proved useful in this case study. However, for the traceability branch starting at the functional model and ending a the low level model, the sequence of refinements did not comply with the correctness criteria from Section 6.2.2. This situation occurred because the functional model didn't have quite the same structure as the tasking model.

In terms of functional analysis, the majority of the completeness and consistency results were preserved through the chain of models. This situation occurred because most of the machines used in the low level model were taken verbatim from the tasking model and from the functional model. The functional analysis performed on the low level UPPAAL model started to stretch the performance of the UPPAAL engine. Some of the verification queries took over 30 minutes to complete and consumed over 500 MB of memory on a Pentium 4 operating at 2.4GHz with 512 MB of memory. This situation became slightly worse for execution time analysis with the addition of the observer automaton and the iterative nature of the analysis algorithm. Nevertheless, the analysis was completely automated, meaning that the UPPAAL verifier could be left alone to explore the state space. Using a mature verification engine like UPPAAL proves practical because the UPPAAL verifier contains various options for how the state space is explored [156]. For example, there are different state space reduction techniques which can be used to obtain better performance, at the cost of the exactness of solutions. Depending on the property to be verified, an overapproximation or an underapproximation of the solution might be acceptable if an exact solution is not feasible given the problem size.

Because the ETC is an industrial application, the implementation of the functionality and the tasking model is not overly complex. Consequently, the amount of resources consumed is rather limited. The resource consumption analysis performed on the low level ETC model did not yield terribly insightful results, but was useful to validate the resource analysis approach. Overall, the ETC case study was an interesting application to study using the framework because it exercised the majority of the components of the framework, while stretching the performance limits of the analysis algorithms. The case study also demonstrated the versatility of the TASM modeling language as the language was used to model a tasking structure, a functional model, and a combination of the two.

8.5 The Timeliner Script Executor: Plant Control System

The Timeliner system has been developed as a scripting environment to automate procedural tasks typically performed by human operators [61]. The system is composed of a high-level input language, a compiler, a run-time system, and a user interface. The Timeliner system is described in details in Section 2.8.3.

The plant control system is a Timeliner application that contains a script used to regulate the cabin temperature and the cabin humidity level of a plant, as shown in Figure 2-12. The Timeliner script uses three actuators to regulate the cabin environment – a heating system, a cooling system, and a humidifier system [238]. The control script is composed of two sequences. The first sequence, called `TEMP_MONITOR`, shown in Listing 8.23, is used to maintain the temperature of the cabin between 20 and 25 Celsius degrees. The second sequence, called `HUMIDITY_MONITOR`, shown in Listing 8.24, is used to maintain the humidity of the cabin between 40 and 60 percent. When the temperature is greater than or equal to 26 Celsius degrees, the sequence starts the cooling system. When the temperature is lower than or equal to 19 Celsius degrees, the sequence starts the heating system. In both instances, the sequence will wait for the temperature to reach an acceptable level before continuing. When the cabin humidity is over 60 percent, the `HUMIDITY_MONITOR` sequence starts the cooling system. When the cabin humidity is lower than 40 percent, the `HUMIDITY_MONITOR` sequence starts the humidifier system. The variable TRYING_TO_COOL_SYSTEM is used to notify the HUMIDITY_MONITOR sequence not to turn off the cooling system if the TEMP_MONITOR sequence is cooling the cabin. The HUMIDITY_MONITOR sequence uses the cooling system to reduce the humidity of the cabin and shares usage of the cooling system with the TEMP_MONITOR sequence.

Listing 8.23 Timeliner TEMP_MONITOR sequence [238]

```
SEQUENCE TEMP_MONITOR
  EVERY 1
    IF TEMPERATURE >= 26 THEN
      SET TRYING_TO_COOL_SYSTEM TO TRUE
      COMMAND COOLING, NEW_STATE=>ON
      WHEN TEMPERATURE <= 22
        SET TRYING_TO_COOL_SYSTEM TO FALSE
        COMMAND COOLING, NEW_STATE=>OFF
      END WHEN
    END IF
    IF TEMPERATURE <= 19 THEN
      COMMAND HEATING, NEW_STATE=>ON
      WHEN TEMPERATURE >= 22
        COMMAND HEATING, NEW_STATE=>OFF
      END WHEN
    END IF
  END EVERY
CLOSE SEQUENCE
```

Listing 8.24 Timeliner HUMID_MONITOR sequence [238]

```
SEQUENCE HUMID_MONITOR
  EVERY 1
    IF HUMIDITY >= 61 THEN
      COMMAND COOLING, NEW_STATE=>ON
      WHEN HUMIDITY <= 50
        IF NOT TRYING_TO_COOL_SYSTEM
          COMMAND COOLING, NEW_STATE=>OFF
        END IF
      END WHEN
    END IF
    IF HUMIDITY <= 39 THEN
      COMMAND HUMIDIFIER, NEW_STATE=>ON
      WHEN HUMIDITY >= 50
        COMMAND HUMIDIFIER, NEW_STATE=>OFF
      END WHEN
    END IF
  END EVERY
CLOSE SEQUENCE
```

The plant control system case study is an interesting case study because it involves the modeling of a software program with time annotations for the program. This case study differs from the production cell case study where the software actions are assumed to be instantaneous and also differs from the electronic throttle controller case study where the timing of actions is contained in the tasking model.

8.5.1 Model

The execution times of various Timeliner statements have been heavily studied by the Charles Stark Draper Laboratory, with timing results documented in [62]. The measures were performed using the Timeliner Testbed, with version CI_024 of the Timeliner Executor, using an embedded real-time 16MHz Intel 80836sx VME board with an 80387 floating point coprocessor. The execution times contained in document [62] are used to model the scripts in the TASM language. The scripts are modeled in the TASM language for the sake of performing timing analysis and to generate test cases. More specifically, the static timing analysis of Timeliner scripts provides execution time guarantees. Traditionally, the Best-Case Execution Time (BCET) and Worst-Case Execution Time (WCET) of one *pass* of a Timeliner script were obtained through manual analysis and through systematic testing. By modeling the script in TASM and using the *iterative bounded liveness* approach presented in Section 5.3, exact values of the BCET and WCET can be obtained automatically.

To model the scripts in the TASM language, the scripts are augmented with labels where the statements contained in the script either block or branch. For example, the `IF` statement is a branching statement; the `WHEN` statement is a blocking statement which will block until its condition becomes true. The structural view of both augmented sequences is shown in Listing 8.25. The possible executions of the sequence can be illustrated using the labels. For example, under nominal conditions, the `TEMP_MONITOR` sequence will execute through the sequence of labels b_0, b_1, b_3, b_0 in one pass. At the next pass, if the temperature is greater than or equal to 26 Celsius degrees, the sequence will execute through the sequence of labels b_0, b_1, b_2 in one pass. Eventually, the sequence will leave b_2 when the temperature is below 22

Celsius degrees. Until that condition is met, the sequence will stay at label b_2 at each pass.

Listing 8.25 Labeled TEMP_MONITOR sequence

```
b0:  EVERY 1

b1:   IF TEMPERATURE >= 26 THEN

b2:      WHEN TEMPERATURE <= 22

b3:   IF TEMPERATURE <= 19 THEN

b4:     WHEN TEMPERATURE >= 22
```

Listing 8.26 Labeled HUMID_MONITOR sequence

```
c0:  EVERY 1

c1:    IF HUMIDITY >= 61 THEN

c2:      WHEN HUMIDITY <= 50

c3:        IF NOT TRYING_TO_COOL_SYSTEM

c4:    IF HUMIDITY <= 39 THEN

c5:      WHEN HUMIDITY >= 50
```

The duration of one step transitions from one label to another label have been calculated using the execution times documented in [62]. The results are shown in Table 8.25 for the `TEMP_MONITOR` sequence and in Table 8.26 for the `HUMIDITY_MONITOR` sequence. Entries in the table marked with a "-" denote that the transition is not possible in one step. It is important to remember that multiple transitions can be taken in one pass, until a blocking statement is reached.

The time of the transitions are used to build the TASM model. The transitions are naturally encoded into rules, and the durations from Table 8.30 and Table 8.31 become the durations of the rule executions.

	b_0	b_1	b_2	b_3	b_4
b_0	685	685	-	-	-
b_1	-	-	2285	1730	-
b_2	-	-	1625	3725	-
b_3	1950	-	-	-	2390
b_4	1630	-	-	-	3195

Table 8.30: Duration of transitions (in μs) between labels of Listing 8.25

	c_0	c_1	c_2	c_3	c_4	c_5
c_0	685	685	-	-	-	-
c_1	-	-	2395	-	1730	-
c_2	-	-	1625	1625	-	-
c_3	-	-	-	-	2160	-
c_4	1950	-	-	-	-	2390
c_5	3195	-	-	-	-	1630

Table 8.31: Duration of transitions (in μs) between labels of Listing 8.26

The Environment

The labels introduced in Listing 8.25 and in Listing 8.26 are encoded in the TASM model in the type Temp_Sequence_Block and in the type Humid_Sequence_Block. The types are used as a program counter to keep track of the location of each sequence through each pass of execution. The Processor_Status type is used to allocate processor usage to Timeliner and to the control task. The TASM model contains only 1 bundle, the plantsim bundle, which contains 2 sequences. Other bundles and sequences can be easily added to the model by extending the Bundle and Sequence types. The list of types is shown in Listing 8.27.

Listing 8.27 User-defined types of the model

```
Status                  := {active, inactive};
Device                  := {on, off};
Temp_Sequence_Block     := {b0, b1, b2, b3, b4};
Humid_Sequence_Block    := {c0, c1, c2, c3, c4, c5);
Processor_Status        := {timeliner, controltask};
Execution_Status        := {done, not_done};
Bundle                  := {plantsim};
Sequence                := {temp_monitor, humid_monitor};
```

The variables that make up the environment contain status variables for the bundles and sequences, the program counters for each sequence, which task has control of the processor, and whether a given pass is done or not done. The environment variables are shown in Appendix F, in Listing F.2.

Timeliner Bundles and Sequences

The execution semantics of the Timeliner language are such that only active bundles are executed and only active sequences are executed, per the organization of Figure 2-11. Given the hierarchical nature of the execution semantics, sub machines are a natural fit to describe this behavior. The sub machine *EXECUTE_BUNDLES* executes all bundles in sequential order. The sub machine *PLANTSIM_BUNDLE* executes all sequences in its bundle, in sequential order, through the *EXECUTE_PLANTSIM_SEQUENCES* sub machine, if the bundle is active. The rules of the *PLANTSIM_BUNDLE* sub machine are shown in Listing 8.28, where the active status of the bundle dictates whether or not the sequences are executed.

Listing 8.28 Rules of the *PLANTSIM_BUNDLE* sub machine

```
R1: Bundle Active
{
  if plantsim_bundle_status = active then
    EXECUTE_PLANTSIM_SEQUENCES();
}

R2: Bundle Inactive
{
  if plantsim_bundle_status = inactive then
    plantism_s := done;
}
```

Each sequence is also wrapped into a sub machine so that the sequences are executed only if they are active. The *SEQUENCE_TEMP_MONITOR* sub machine and the *SEQUENCE_HUMID_MONITOR* sub machine are wrapper sub machines that will execute the work of the sequences only if the given sequence is active. The *SEQUENCE_TEMP_MONITOR* is shown in Listing 8.29.

Listing 8.29 Rules of the *SEQUENCE_TEMP_MONITOR* sub machine

```
R1: Sequence Active
{
  if temp_seq_status = active then
    SEQUENCE_TEMP_MONITOR_WORK();
}

R2: Sequence Inactive
{
  if temp_seq_status = inactive then
    temp_seq_s := done;
}
```

TEMP_MONITOR Sequence

The *SEQUENCE_TEMP_MONITOR_WORK* sub machine describes the logic of the temperature monitor sequence. Listing 8.30 shows 2 of the 11 rules that make up the machine. Each rule represents a durative transition between the sequence labels, as shown in Table 8.30. The names of each rule are of the form "`bi -> bj`" to indicate that the rule describes the transition between the label b_i to the label b_j. For example, rule R_5 of Listing 8.30 describes the transition between label b_2 and label b_3. Referring to Listing 8.23 and Listing 8.25, the rule is fired only if the temperature is less than or equal to 22 Celsius degrees. If the temperature is above 22 Celsius degrees, the sequence will stay on the WHEN statement until the temperature falls below 22 Celsius degrees. This behavior is captured by rule R_4. The time annotations of each rule are congruent to the calculations summarized in Table 8.30. The complete list of rules of the *SEQUENCE_TEMP_MONITOR_WORK* are given in Listing F.12 and in Listing F.13.

Listing 8.30 Rules of the *SEQUENCE_TEMP_MONITOR_WORK* sub machine (partial)

```
R4: b2 -> b2
{
  t := 1625;

  if temp_seq_b = b2 and temperature > 22 then
    temp_seq_b := b2;
    temp_seq_s := done;
}

R5: b2 -> b3
{
  t := 1730;

  if temp_seq_b = b2 and temperature <= 22 then
    temp_seq_b            := b3;
    trying_to_cool_system := False;
    cooling               := off;
}
```

The HUMIDITY_MONITOR sequence is built analogously to the TEMP_MONITOR sequence through the *SEQUENCE_HUMIDITY_MONITOR* sub machine and the *SEQUENCE_HUMIDITY_MONITOR_WORK* sub machine. The structure of the two sequences is identical with regards to the status of the sequence and the list of rules is built based on Table 8.31.

Control Task

In order to model the dynamics of script execution, a separate "control task" is modeled, which shares the processor with Timeliner. The control task is modeled in

TASM as a main machine which acts as a "dummy" task which performs no function other than consume time. The execution time of the control task is set between 3500 μs and 5000 μs. The rules of the *Control_Task* main machine are shown in Listing 8.31.

Listing 8.31 Rules of the *Control_Task* main machine

```
R1: Control Task
{
  t := [3500, 5000];

  if processor = controltask and execution = not_done then
    execution := done;
}

R2: Else
{
  t := next;

  else then
    skip;
}
```

Scheduler

The scheduler is the component responsible for switching processor control between Timeliner and the control task. The processor is allocated to the two competing tasks on a round-robin basis. The scheduler waits for the task that owns the processor to signal that it has finished execution for the round. In terms of Timeliner, a round of execution corresponds to a pass. The scheduler is modeled without preemption in order to verify the maximum execution time of a single pass of Timeliner. The scheduler uses the `processor` variable and the `execution` variable to determine which task has control of the processor. The context switching time is assumed to be 1000 μs. The functionality is wrapped in a main machine, called *Scheduler.* The rules of this main machine are shown in Listing 8.32.

The Plant Cabin - Temperature and Humidity

The behavior of the humidity and the temperature inside the plant cabin influences the Timeliner sequence execution. For example, if the temperature is below 20 Celsius

Listing 8.32 Rules of the *Scheduler* main machine

```
R1: Controller
{
  t := 1000;

  if processor = controltask and execution = done then
    processor := timeliner;
    execution := not_done;
}

R2: Timeliner
{
  t := 1000;

  if processor = timeliner and execution = done then
    processor := controltask;
    execution := not_done;
}

R3: Else
{
  t := next;

  else then
    skip;
}
```

degrees, the TEMPERATURE_MONITOR sequence will turn on the heating system in order to warm the cabin. In order to verify all the potential behaviors of the Timeliner script, the behavior affecting the execution of the script must be modeled. The behavior of the environment is non-deterministic and it is assumed that the temperature and humidity can vacillate unexpectedly. The restrictions on the vacillation of the atmosphere are such that the temperature and humidity will not vary unexpectedly if any of the actuating systems, namely heating, cooling, and humidifier are turned on. The behavior of the atmosphere is encoded in two main machines to represent the behavior of the two environment variables. The non-deterministic behavior of the `temperature` variable is encoded in a main machine called *Temperature*. Three of the rules of the machines are shown in Listing 8.33.In Listing 8.33, since the guards of the first two rules are identical, when both guards are enabled, one of the rules is selected non-deterministically and is executed. The first rule, rule R_1, describes nominal behavior, where the temperature does not change. The second rule, rule R_2, switches the temperature from nominal to "too low" per the behavior of the Timeliner script. The time annotations are selected to ensure that all the behaviors of the script are exercised. The time annotation of 685 μs represents the smallest step in each sequence. The third rule shown in Listing 8.33, rule R_5, describes the behavior of the temperature variable when the heating system is turned on to remedy a drop in temperature below the critical threshold. The duration of the interval to return the temperature to nominal is between 0 and 1500 μs. This annotation was selected to exercise the different behaviors of the script. For example, an instantaneous return to nominal will cause the script to execute all the sequences in a pass. If the duration to return the temperature to nominal is higher than 685 μs, Timeliner will block on the WHEN statement, while waiting for the temperature to return to the nominal value. Furthermore, the upper bound on the duration, 1500 μs ensures that it is possible for the script to execute on pass where the only statement in the pass is the execution of the blocking WHEN statement. A value higher than 1500 μs would cause the script to execute multiple passes where the script executes only the blocking WHEN statement. This situation would be redundant from an analysis and simulation perspective since it would not exercise different script behavior. The behavior of the humidity variable is encoded in an analogous main machine called *Humidity*.

The model of the temperature and humidity variables is simplistic and does not adequately reflect the differential equation relationship between humidity, temperature, and the heating, cooling, and humidifier. However, the goal of the model is to analyze the behavior of the Timeliner script and the model of the temperature and humidity is sufficient to exercise the necessary behavior of the script.

Listing 8.33 Rules of the *Temperature* main machine (partial)

```
R1:
{
  t := 685;

  if temperature > 19 and temperature < 26 and
     cooling = off and heating = off and humidifier = off then
    skip;
}

R2:
{
  t := 685;

  if temperature > 19 and temperature < 26 and
     cooling = off and heating = off and humidifier = off then
    temperature      := 18;
}

R5: Heating on
{
  t := [0, 1500];

  if temperature < 20 and heating = on and cooling = off then
    temperature := 24;
}
```

Complete Model

The complete TASM model contains 5 main machines, no function machines, and 7 sub machines. The complete Timeliner plant control system model is documented in Appendix F where the list of all machines is provided in Table F.1.

UPPAAL Model

The TASM model is translated to UPPAAL to perform model checking of safety and liveness properties and to perform execution time analysis. The first step of the translation to the timed automata of UPPAAL removes hierarchical composition from the main machines. In the TASM model, only the *Timeliner* main machine uses hierarchical composition, as it uses 7 sub machines. The flattened version of the *Timeliner* main machine contains 27 rules. The complete UPPAAL model yields 10 timed automata – 1 automaton for the *Timeliner* script and execution, 1 automaton for the scheduler, 1 automaton for the control task, 3 automata to enforce the urgent channel transitions of the 3 aforementioned machines, 2 automata to describe the evolution of the `temperature` and `humidity` variables (the environment). The timed automaton for the scheduler machine is shown in Figure 8-9. In the scheduler automaton, the `processor` variable is used to denote whether the control task (`processor == 2`) or Timeliner (`processor == 1`) has control of the processor. The variable `execution` is used to denote whether the component currently using the processor has completed (`execution == 1`) or not (`execution == 2`). The variable `Scheduler_else` is an urgent channel that is used to keep the scheduler waiting in the *Scheduler_ELSE* location until it needs to switch the processor context. The *pivot* location is used to either perform an action, if applicable, or to move to the *Scheduler_ELSE* location if no action is enabled. The *Scheduler_ELSE* location is the only location where time can elapse because of actions of other automata.

The most complex automaton of the translated model is the automaton describing the behavior of the *Timeliner* main machine. The complete automaton contains 27 locations and 54 transitions, and retains the general form of the Scheduler automaton shown in Figure 8-9. As for the automaton in Figure 8-9, all locations have invariants, except for the *Scheduler_ELSE* location, which is guarded by an urgent channel called *Scheduler_else*.

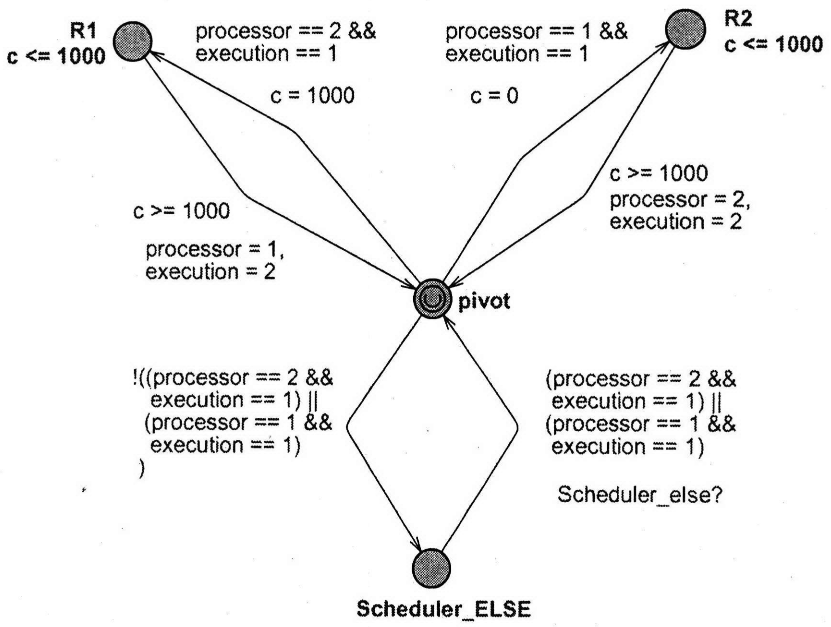


Figure 8-9: Automaton for the Scheduler main machine

8.5.2 Functional Analysis

The UPPAAL model can be used to verify safety and liveness properties of the Timeliner script. Safety properties that can be verified include the mutual exclusion of the different actuators:

- The cooling system and the humidifier should not be on simultaneously
- The heating system and the cooling system should not be on simultaneously

If any of these properties hold, they signal a conflict between the logic in the two scripts. The reason why these situations are undesirable is because these two systems contradict each other and hence would cancel the desired effect. If such a situation occurs, both sequences will end up waiting forever for the environment to change, a situation that can't happen under these circumstances, which would result in the system being deadlocked. These two safety properties can be translated into the temporal logic query language of UPPAAL :

- `A[] !(cooling == 1 && humidifier == 1)`
- `A[] !(heating == 1 && cooling == 1)`

Executing the first query in the UPPAAL verifier shows that the safety property holds. However, executing the second property in the UPPAAL verifier unveils that the property does not hold in the model. In the model, it is possible for the humidity to be over 60% and for the temperature to be under 20 Celsius degrees. In this situation, the `TEMPERATURE_MONITOR` sequence would turn on the heating system to raise the temperature. In turn, the `HUMIDITY_MONITOR` sequence would turn on the cooling system to reduce the ambient humidity. In the model, this case does not result in the heating and cooling system canceling their effects because the humidity variable is unaffected by the heating system. When the heating system and the cooling system are on simultaneously, the humidity would decrease while the temperature stays constant. Once the humidity reaches the nominal level, the `HUMIDITY_MONITOR` sequence will turn off the cooling system, at which point the cabin temperature will start to rise and eventually reach a nominal level. Since this safety property does not hold in the model, it raises an interesting question about whether this situation is a reachable state in the environment. Perhaps there are unstated assumptions about the behavior of the temperature and the humidity variables.

The liveness properties of the Timeliner script which can be verified include the behavior of the script in the face on off-nominal values of the cabin variables:

- If the temperature falls below 20 Celsius degrees, the heating system will eventually be turned on
- If the temperature rises over 25 Celsius degrees, the cooling system will eventually be turned on
- If the humidity falls below 40%, the humidifier system will eventually be turned on
- If the humidity rises over 60%, the cooling system will eventually be turned on

All of these properties are expected to hold because they describe the basic requirements of the monitor sequences. These properties can be stated using the temporal logic query language of UPPAAL :

- `(temperature < 19) --> (heating == 1)`
- `(temperature > 26) --> (cooling == 1)`
- `(humidity < 39) --> (humidifier == 1)`
- `(humidity > 61) --> (cooling == 1)`

These properties can be run through the UPPAAL verifier and can be shown to hold for the given model. Other liveness properties can also be verified, to ensure that the model behaves as expected, as needed. For example, it can be verified that if the temperature is below 20 Celsius degrees and the heating system is on, the temperature will eventually return to the nominal level.

Completeness and Consistency

The analysis of completeness and consistency was performed using the approach described in Section 5.1. The results of verifying completeness are shown in Table 8.32. The table shows, for each machine, the number of propositions, the number of clauses, and whether or not the machine is complete. For machines that are trivially complete, the number of propositions and clauses is listed as "N/A". A similar table, Table 8.33, presents the results of verifying the consistency of each machine.

Name	Propositions	Clauses	Complete
Control_Task	N/A	N/A	Yes
Scheduler	N/A	N/A	Yes
Timeliner	N/A	N/A	Yes
Temperature	N/A	N/A	Yes
Humidity	N/A	N/A	Yes
EXECUTE_BUNDLES	3	5	Yes
PLANTSIM_BUNDLE	2	4	Yes
EXECUTE_PLANTSIM_SEQUENCES	6	10	Yes
SEQUENCE_TEMP_MONITOR	2	4	Yes
SEQUENCE_TEMP_MONITOR_WORK	10	34	Yes
SEQUENCE_HUMID_MONITOR	2	4	Yes
SEQUENCE_HUMID_MONITOR_WORK	11	46	Yes

Table 8.32: Completeness analysis results for the Timeliner plant control system

Name	Propositions	Clauses	Consistent
Control_Task	N/A	N/A	Yes
Scheduler	4	8	Yes
Timeliner	N/A	N/A	Yes
Temperature	9	25	No
Humidity	9	25	No
EXECUTE_BUNDLES	3	7	Yes
PLANTSIM_BUNDLE	2	4	Yes
EXECUTE_PLANTSIM_SEQUENCES	6	10	Yes
SEQUENCE_TEMP_MONITOR	2	4	Yes
SEQUENCE_TEMP_MONITOR_WORK	10	34	Yes
SEQUENCE_HUMID_MONITOR	2	4	Yes
SEQUENCE_HUMID_MONITOR_WORK	11	46	Yes

Table 8.33: Consistency analysis results for the Timeliner plant control system

8.5.3 Execution Time Analysis

The execution time of the model is analyzed using the approach described in Section 5.3. The first goal is to verify the WCET and BCET for *one pass* of the Timeliner script executor. In the model, the beginning of one pass of the script executor begins with the state "`processor = timeliner, execution = not_done`". Consequently, the state predicate for the beginning of the path, p_0, is the UPPAAL state `(processor == 1 && execution == 2)`. Conversely, the state predicate for the final state of the path, p_1, is the UPPAAL state `(processor == 1 && execution == 1)`, which states that Timeliner still has control of the processor, but that its execution is done for the current pass. Consequently, the goal is to analyze the execution time of the path `(processor == 1 && execution == 2)` $\mapsto$ `(processor == 1 && execution == 2)`. The observer automaton to measure the execution time of one pass of Timeliner is shown in Figure 8-10. Using ϕ_{init}, the trace stored in the simulator for the observer automaton yields $t_{init} = 11030\mu s$. In terms of the Timeliner script, this trace corresponds to execution of labels b_0, b_1, b_2 in one pass for the `TEMP_MONITOR` sequence, followed by execution of labels c_0, c_1, c_4 for the `HUMID_MONITOR` sequence. t_{init} is used to iteratively establish the satisfiability of ϕ_{max}.

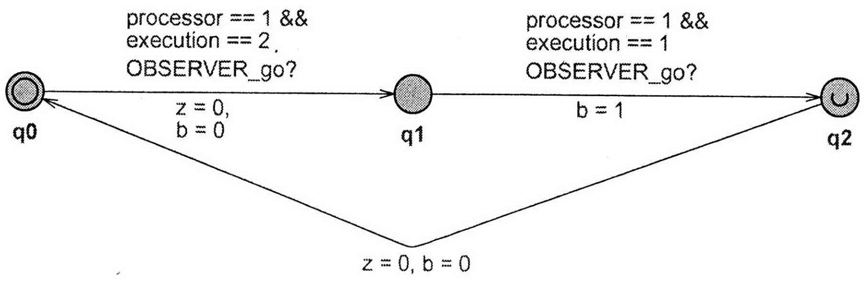


Figure 8-10: Observer automaton to measure the execution time of one pass of Timeliner, for the scripts of Listing 8.23 and of Listing 8.24

Using the approach explained in Section 5.3, the WCET, $t_{max_{p_0 \mapsto p_1}}$, was established to be $16815\mu s$ after 4 iterations. The iterations are shown in Table 8.34. The behavior corresponding to the WCET of one pass is the sequence of execution of labels b_0, b_1, b_3, b_4, b_0 in one pass for the `TEMP_MONITOR` sequence, followed by execution of labels $c_0, c_1, c_2, c_3, c_4, c_0$ for the `HUMID_MONITOR` sequence.

Iteration	Time
0	11030μs
1	12870μs
2	14435μs
3	15250μs
4	16815μs

Table 8.34: Execution time analysis results for the WCET of one pass of Timeliner

To establish the BCET, $t_{min_{p_0 \mapsto p_1}}$, a similar iterative strategy is used using ϕ_{min}. $t_{min_{p_0 \mapsto p_1}}$ is established to be $3255\mu s$ after 2 iterations. The iterations are shown in Table 8.35. The trace which yields the BCET of one pass of Timeliner is the simple case of executing a single statement in each sequence. This trace corresponds to the execution of one blocking **WHEN** statement for the **TEMP_MONITOR** sequence, and the execution of one blocking **WHEN** statement for the **HUMID_MONITOR** sequence. One example displaying the BCET of one pass is the execution of label b_2 in the **TEMP_MONITOR** sequence followed by the execution of label c_2 in the **HUMID_MONITOR** sequence.

Iteration	Time
0	11030μs
1	8960μs
2	3255μs

Table 8.35: Execution time analysis results for the BCET of one pass of Timeliner

End-to-End Latency

End-to-end latency refers to the amount of time that it takes for the system to react to a change in the environment. In the context of the plant control system, one example of end-to-end latency involves the amount of time required for the plant control system to turn on the heating system when the temperature falls under 20 Celsius degrees. In terms of a predicate over state variables, this behavior corresponds to the path between a state where "`temperature <= 19, heating = off`" to a state where "`heating = on`". The observer automaton to verify this property is shown in Figure 8-11. The WCET of the end-to-end latency corresponding to this automaton is

$22570\mu s$, reached after 2 iterations. The BCET of the end-to-end latency corresponding to this automaton is $2390\mu s$. Since this property is a system property, the time used in the calculation of the WCET and the BCET involves delays in the scheduler and the execution time of the control task. The behavior corresponding to the WCET is the behavior where the temperature drop occurs after rule R_6 has just started executing in the *TEMP_MONITOR_SEQUENCE_WORK* sub machine. The WCET is calculated using rule R_6 ($1950\mu s$) of the *TEMP_MONITOR_SEQUENCE_WORK* sub machine, followed by rules R_1 ($685\mu s$), R_2 ($2395\mu s$), R_5 ($1625\mu s$), R_6 ($2160\mu s$), R_9 ($1950\mu s$) of the *HUMID_MONITOR_SEQUENCE_WORK* sub machine, followed by a context-switch by the scheduler ($1000\mu s$), the maximum execution time of the control task ($5000\mu s$), a context switch by the scheduler ($1000\mu s$), and rules R_1 ($685\mu s$), R_3 ($1730\mu s$), R_7 ($2390\mu s$) of the *TEMP_MONITOR_SEQUENCE_WORK* sub machine. In terms of Listing 8.23 and Listing 8.24, the maximum end-to-end latency will occur when the temperature drop occurs when the `TEMP_MONITOR` sequence has started executing label b_3, the `HUMID_MONITOR` sequence is at label c_0, and the humidity is over 60%. For the minimum end-to-end latency, the value $2390\mu s$ corresponds to execution of rule R_7 of the *TEMP_MONITOR_SEQUENCE_WORK* sub machine. This means that the temperature drop happens just before the rule begins executing. In terms of Listing 8.23, this corresponds to the temperature drop occurring just before the sequence is at label b_3.

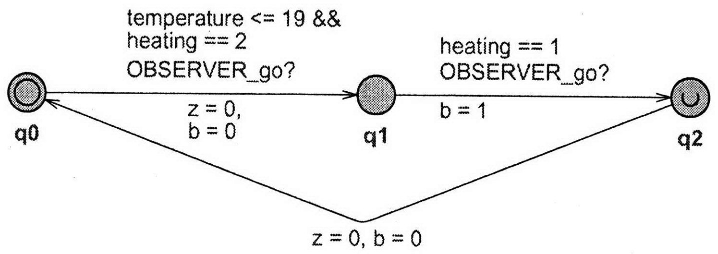


Figure 8-11: Observer automaton to measure the end-to-end latency of Timeliner for a temperature drop, for the scripts of Listing 8.23 and of Listing 8.24

Similar observers were formulated for the other conditions leading to the heating, cooling, and humidifier systems being turned on. The end-to-end latency results, alongside the number of iterations needed to converge on the values are reported in

Table 8.36.

p_0	p_1	WCET	Iterations	BCET	Iterations
heating = off, temperature < 20	heating = on	11030μs	3	2390μs	10
cooling = off, temperature > 25, humidity >= 40, humidity <= 60	cooling = on	21650μs	4	2285μs	8
cooling = off, humidity > 60, temperature >= 20, temperature <= 25	cooling = on	21760μs	6	2285μs	7
humidifier = off, humidity < 40	humidifier = on	21755μs	6	2390μs	18

Table 8.36: End-to-end latency analysis results for the plant control system

Since both sequences share the usage of the cooling system, extra conditions need to be added to p_0. Otherwise, the BCET would be trivially 0. For example, if the end-to-end latency of the Timeliner script turning on the cooling system, to remedy a rise in temperature, is to be measured, the condition that the humidity be nominal at the beginning of the path is necessary. Otherwise, the BCET would result when the rise in temperature occurs right after the HUMID_MONITOR sequence has turned on the cooling system to remedy a rise in humidity.

8.5.4 Test Case Generation

In the Timeliner case study, the *Temperature* and the *Humidity* main machines are used to simulate environmental behavior and are not included in the generation of test cases. Consequently, test cases are generated for all other machines , using the algorithm described in Listing 7.8. The results of the test case generation are shown in Table 8.37. In Table 8.37, the first column provides the machine name, the second column lists the number of test case templates in the test suite template for the machine, and the third column lists the number of test cases from the test suite required for unit testing of the machine. Per the approach described in Chapter 7, the test cases are generated using the rule coverage criterion, explained in Section 7.1.1.

Machine	Test Suite	Unit Testing
Control_Task	2	2
Scheduler	3	3
Timeliner	27	2
EXECUTE_BUNDLES	26	2
PLANTSIM_BUNDLE	25	2
EXECUTE_PLANTSIM_SEQUENCES	24	4
SEQUENCE_TEMP_MONITOR	10	2
SEQUENCE_TEMP_MONITOR_WORK	9	9
SEQUENCE_HUMIDITY_MONITOR	12	2
SEQUENCE_HUMIDITY_MONITOR_WORK	11	11

Table 8.37: Test case generation results for the plant control system

Pre State	Post State	Coverage Item
$processor\{timeliner\}$, $execution\{not\_done\}$, $exec\_bundle\{plantsim\}$, $plantsim\_bundle\_status\{active\}$, $exec\_seq\{humid\_monitor\}$, $humid\_seq\_s\{not\_done\}$, $humid\_seq\_b\{c1\}$, $humidity\{[61, 100]\}$	$humid\_seq\_b\{c2\}$, $cooling\{on\}$	$Timeliner.R_1$, $EXECUTE\_BUNDLES.R_1$, $PLANTSIM\_BUNDLE.R_1$, $EXECUTE\_PLANTSIM-\_SEQUENCES.R_3$, $SEQUENCE\_HUMIDITY-\_MONITOR.R_2$

Table 8.38: Sample test case template from the test suite for the *Timeliner* main machine

A sample test case template from the *Timeliner* main machine test suite is shown in Table 8.38.

8.5.5 Discussion

The Timeliner plant control system case study focused on execution time analysis for various properties of the Timeliner script. More specifically, the analysis performed enabled the calculation of the minimum and maximum execution times for a single pass of the Timeliner script. Furthermore, end-to-end latency analysis uncovered the worst case and best case response time that the system can provide for a change in the environment. The analysis was performed for four different conditions that affect the execution of the script. The interesting point to note is that the execution time analysis was performed automatically and without modifying the generated UPPAAL model. The use of observer automata, as described in Section 5.3, provides a generic and unintrusive means of analyzing the system model. This case study modeled an example at the detailed software level, using lab measurements for various statements of the language. The modeling and analysis of Timeliner scripts could certainly be generalized to support the entire Timeliner language.

The functional analysis performed on the plant control system involved verifying safety properties and a different type of liveness property than in other case studies. The "ϕ_1 --> ϕ_2" UPPAAL path quantifier states that, in all paths, if ϕ_1 holds at some point for a state in the path, then ϕ_2 will eventually hold for a later state in the path. This type of property is important to determine that the script achieves its basic functionality, such as turning on the heating system after a temperature drop. This property is different than a simple reachability property which can state that a particular state is reachable is some path, or in all paths. By using the "-->" operator of UPPAAL , the properties of sequences of states in all paths can be formulated. As in other case studies, the analysis of consistency and completeness proved useful during model development and model debugging, but the end results of the analysis are not terribly insightful. Nevertheless, the plant control system case study proved to be a case study of manageable size to illustrate in details the capabilities of the framework in terms of execution time analysis and test case generation.

8.6 Segue into Chapter 9

This chapter presented three case studies used to evaluate the capabilities of the proposed framework and the TASM language. The case studies were used to illustrate the modeling facilities of the language, the analysis capabilities of the framework, the traceability approach, and the test case generation strategy. In the following chapter, Chapter 9, the contributions of the presented research are reviewed in light of the presented framework and the results of the case studies. Chapter 9 also provides guidance for future work related to the presented research.

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 9

Conclusion

This chapter reexamines the objectives of the research, described in Chapter 1, in light of the presented framework and the experimental results obtained through the case studies. The research contributions of the thesis are also revisited in light of the information presented in the previous chapters. This chapter also provides a synthesized critique of the various components of the presented framework and describes additional developments that could be achieved as part of future research.

9.1 Research Objectives and Contributions

The research objectives of the presented thesis were first described in Chapter 1. The objectives sought to address key challenges in the engineering of embedded real-time systems. These challenges, alongside proposed solutions, are repeated below:

- The high complexity of modern software systems by providing a model-based approach to software-intensive system engineering.
- The high cost of Verification and Validation (V & V) activities by leveraging the use of models to automate engineering activities.
- The challenges in using formal methods in an engineering context by providing a novel literate specification language and hiding verification details in a push-button approach.

- The lack of integration between models by providing bi-directional traceability across levels of abstraction.

- The lack of integration of the state-of-the art in individual disciplines by providing an overarching engineering framework.

Certainly, the framework presented in this thesis was flexible enough to model and analyze archetypical case studies from the embedded real-time system domain. The case studies concerned applications from the manufacturing industry, the automotive industry, and the aerospace industry. Furthermore, the framework proved versatile enough to model and study the three key aspects of the systems of interest – function, time, and resources. The framework succeeded in tackling some of the complexity issues of these systems by providing a building block approach to system construction and analysis. The high cost of V & V activities was addressed by reusing well-established analysis engines that provided automated analysis, namely the UPPAAL tool suite and the SAT4J *SAT* solver, and by performing verification and test case generation activities automatically. The challenges of using formal methods in an engineering context was addressed in two ways. The first approach built on the success of Abstract State Machines (ASM), which have been used successfully in an engineering context [41]. Furthermore, the TASM toolset provides an integrated tool suite that hides many of the details involved in the translation and the verification using the analysis engines. The experience in the Embedded System Laboratory, at conferences, and through outside evaluation has been positive among the engineering community. The integration of models has been addressed through the bi-directional traceability strategy, which has proved to be versatile enough to handle the incorporation of seemingly disparate models, such as a tasking model and a functional model. The presented framework incorporates ideas from a number of communities, including the formal methods community, the test case generation community, and the scheduling community. Incorporating these different disciplines into the framework provides an integrated approach for end-to-end embedded real-time system engineering.

The research contributions of the presented thesis were first described in Chapter 1. These contributions are reiterated below:

- A new specification language for embedded real-time systems, the Timed Abstract State Machine (TASM) language, which extends the theory of Abstract State Machines (ASM). The TASM language integrates the specification of functional and non-functional properties – function, time, and resources.
- A set of verification procedures for automated analysis of models using generally available analysis engines. The analysis procedures include completeness and consistency, execution time, and resource consumption.
- An approach to traceability of system models that incorporates syntactic change and semantic integrity.
- A generic and extensible approach to automatically generate test cases for unit testing, integration testing, and regression testing.
- An integrated framework for modeling, simulation, verification, and test-case generation for embedded real-time systems.
- An integrated toolset implementing the capabilities of the framework

The Timed Abstract State Machine (TASM) language proved a versatile language to express different aspects of embedded real-time systems. Furthermore, the language proved to be a formal yet *literate* specification language, based on the experience using the language. The verification procedures for consistency and completeness improved on previous approaches by formulating the problem in a generic way and providing verification using an existing analysis engine, instead of requiring language-specific algorithms. For execution time analysis, the use of observer automata to formulate the timing properties to be verified proved flexible enough to analyze interesting facets of models such as end-to-end latency, without requiring changes to the model. The provided traceability approach supplies a unique combination of syntactic traceability and semantic equivalence, two concepts typically treated separately. The test case generation strategy introduced the concepts of test case templates to define generic test cases and to provide a scalable and reusable approach to test case generation. The traceability approach was utilized successfully to identify invalidated test cases based on model changes and to generate the necessary test cases to test the changes according to the rule coverage criterion. All of the functionality of the framework is provided in the context of the TASM language, thereby integrating various facets of model building and various engineering functionality, all under the TASM umbrella. Furthermore, the capabilities of the framework were successfully implemented into an integrated toolset, the TASM toolset, to provide end-to-end system engineering tool support, hiding numerous implementation details and removing the need to gain a deep understanding of different languages and tools, such as the UPPAAL tool suite.

9.2 Framework Evaluation

The presented framework was exercised using 3 case studies, as described in Chapter 8. This section provides a "post-mortem" analysis of each feature of the presented research, in light of the experience gained during the case studies. This section summarizes and expands on the discussions following each case study in Chapter 8.

9.2.1 The TASM Language

Based on the case studies, the TASM language proved versatile in modeling a wide variety of system behavior. For example, the language was used successfully to model software and hardware components in the production cell case study. Furthermore, at the software level, the software proved adequate to model software code, as evidenced in the Timeliner case study, control-theoric behavior, and a task/scheduler abstraction, as evidenced in the electronic throttle controller case study. All three facets of the TASM language, namely function, time, and resources, were used successfully to model system behavior in the case studies. The hierarchical composition facilities of the language proved useful to structure and reuse specifications. Furthermore, the composition facilities were instrumental in performing analysis in isolation, as was shown in the electronic throttle controller case study. The use of sub machines in parallel within a step made model construction easier in the case of the production cell case study, but led to exponential growth during the translation to UPPAAL . In terms of usability of the syntax and understanding of the semantics, the language proved relatively easy to learn for someone with a standard programming background. Experience with undergraduate researchers, experience with visiting scientists in the embedded systems laboratory, experience presenting the language at various conferences and workshops, and experience working with engineers from the Charles Stark Draper laboratory all provided evidence that the TASM language is understandable and usable by a user with a standard programming background. These anecdotal results surrounding the use of the TASM language are in line with similar past experiences with ASM [41].

Some aspects of the TASM language were not as easily grasped and were slightly clumsier to use than initially anticipated. The first facet which led to confusion is the parallel nature of update sets within a step. Since the updates happen atomically after the step has finished executing, users were slightly confused that the results of updates which happen sequentially in a rule definition are not available to sub machines or other expressions within the same step. This was especially true for users who were expecting the same semantics as that of a programming language. In certain cases, this situation lead to clumsy specifications, as is the case in the production cell case study, where the arm position is set based on the robot angle. However, situations such as this one were the exception more than the rule and users grasped the importance and subtleties of the step concept with a simple explanation. Another area that led to confusion was the synchronization of parallel main machines with respect to one another. Since the communication between machines happen through shared variables only, wait states had to be devised in the production cell case study and in the electronic throttle controller case study in order to let the environment (other machines) make progress. However, after creating a few models, communication through shared variables became more natural and proved to be adequate. Possible extensions to the TASM language, to enable other synchronization and communication mechanisms, are discussed in Section 9.3.1. Overall, the TASM language proved simple and easy to use and learn, both from a personal experience perspective and from the perspective of peers and colleagues.

9.2.2 Static Analysis

The static analysis capabilities of the framework include functional analysis, execution time analysis, and resource consumption analysis. The functional analysis provided by the framework includes the automated verification of completeness and consistency, and the automated analysis of safety and liveness properties. While the end results of the completeness and consistency analysis did not yield terribly insightful results, the analysis proved indispensable during the model building stage. The verification of functional properties achieved through the UPPAAL verifier was instrumental, both to debug the TASM model and to gain insight into the subtleties of the case studies. The execution time analysis also proved insightful, especially the analysis of end-to-end latency. In the electronic throttle controller case study, the end-to-end latency analysis uncovered an interesting inefficiency in the fault detection strategy. The resource consumption analysis provided an interesting window into the possible parallel behavior of the modeled system.

The bridge between the TASM model and the SAT4J *SAT* solver is quite clear and the counterexample translation is straightforward since it simply maps a Boolean state to a state in the TASM model. However, the bridge between the TASM model and the UPPAAL model is less clear. If a property does not hold in the UPPAAL model, the tool suite returns a path leading to the state which contradicts the property. Since the UPPAAL model is derived using the "flattened" version of the TASM model, the sequence of rule executions wasn't always clear from the generated trace. Furthermore, when a property did not hold in the UPPAAL model, three possibilities were encountered – the TASM model was incorrect, the translation to UPPAAL was incorrect, or the design logic was incorrect. When errors were present in the TASM model, the verification proved insightful to ensure the correctness of the model, especially after simulation scenarios were designed to validate the model. Design logic errors were uncovered using the analysis through UPPAAL although the design errors were often related to the stated property making invalid assumptions about the semantics of the model. For example, in the electronic throttle controller, it is possible to reach a state where the engine revolution is over the critical RPM but the controller is not in the limiting mode. This reachability property is not so much a design error as much as it is a byproduct of the physical reality that there is a delay between an event of interest in the environment and a corrective action by the controller.

9.2.3 Bi-Directional Traceability

The traceability approach presented in Chapter 6 was used on the electronic throttle controller case study in Chapter 8. The benefits of traceability have been established in the software engineering community [217] and the presented approach should provide a usable and flexible means to achieved traceability between models. The theory of refinement has always been intriguing to engineers and mathematicians alike since the seminal paper by Niklaus Wirth [250]. The idea of performing program and system development in a provably correct fashion is an attractive proposition. However, it remains unclear whether such a proposition is realizable in practice since theories of refinement require a fairly strict correspondence between model semantics. In doing so, it is always easier to build models that comply with the refinement theory after the fact instead of building models independently and later try to relate their semantics. Perhaps theories of refinement are best applied in a verification context where the system has been designed and implemented, such as the verification of the Java virtual machine in [237]. Nevertheless, the set of refinement types presented in Section 6.2.1 provide a basis for a disciplined approach to model development. While the correctness criteria presented in Section 6.2.2 seem to apply only to a set of restricted special cases, they proved useful in the electronic throttle controller case study to avoid repeating verification activities unnecessarily, resulting in reduced V & V cost.

9.2.4 Test Case Generation

The test case generation strategy proved fairly straightforward for the case studies. The algorithms presented in Chapter 7 were applied directly without too much trouble. The regression testing strategy was applied to the Electronic Throttle Controller (ETC) case study successfully using the traceability strategy. Unlike the semantic guarantees provided by the correctness criteria, the regression testing approach does not make assumptions about the restrictions of the changes made through the refinement. Consequently, the approach used to regenerate, create, identify, and execute the necessary test cases is applicable at large, where the worst-case corresponds to applying the test case generation algorithm anew.

Since the test case generation is performed in the context of the model, the generated test cases need to be related to a system under test in some way. The ability to apply test cases generated based on the model to the system under test is discussed as possible enhancements to the framework in Section 9.3. Furthermore, when generating test cases, it was assumed that the model could be started in any state and that the effect of executing a single step could be observed at the end of the step. While these assumptions might be valid for a TASM model, they might not be valid for the system under test, which might need to be started in a specific initial state and whose output would not be observable until after a number of steps have been performed. Nevertheless, the test case generation approach provided by the framework provides a strong basis for developing a theory relating the test cases generated by the model to the system under test. As is discussed in Section 7.5.1, the proposed approach can be extended to create test sequences that could be applied to a specific system being tested.

Finally, the coverage criterion used for test case generation is the rule coverage criterion [103]. Rule coverage in the TASM model is analogous to structural coverage for implemented code but weaker than transition coverage or state coverage in finite state automata [102]. More fine grained analysis of the rule guards might be needed to comply with more complex coverage criteria such as the Modified Condition/Decision

Coverage (MD/DC) criterion or with specification-based coverage criteria [143]. Nevertheless, the test case generation strategy provided by the framework presents an intuitive and flexible approach to automatically generate test cases based on TASM models, with acceptable scalability.

9.2.5 Scalability

The topic of scalability is an important one in formal methods and in model-based software engineering [124]. For specification and simulation, scalability does not pose problems since the TASM language provides ample mechanisms for structuring specifications into reusable units using sub machine and function machines. However, if a large number of environment variables are used, the lack of namespaces could present inconvenience to avoid name clashes. In the presented framework, scalability issues arise in the translation of TASM models to the input language of the third party analysis engines. The complexity of the translation algorithm for the mapping to *SAT* is analyzed in Section B.3.1 and the complexity of the mapping to UPPAAL is analyzed in Section C.2.3. For the translation to *SAT*, the use of multiple integer variables in the specification can lead to exponential growth in the size of the Boolean formula. For the translation to the timed automata of UPPAAL, the number of generated automata grows linearly with the number of main machines. However, as evidenced in the production cell case study, the use of multiple sub machine calls within a given rule can lead to exponential growth in the number of locations in the generated timed automata. This occurs because the timed automata of UPPAAL do not contain facilities for hierarchical composition and hence the TASM model needs to be "flattened" by removing the hierarchical composition, essentially taking the cross product of the hierarchical units of composition.

Aside from the scalability of the translation, the static analysis features of the presented framework also lead to scalability issues. *SAT* solving is a well known NP-Complete problem [232], meaning that the performance of the analysis grows exponentially with linear growth of the Boolean formula. For the case of consistency, this problem is exacerbated by the need to verify all pairs of rules against one another.

For the case studies presented in Chapter 8, the analysis of completeness and consistency did not present scalability issues. Nevertheless, the nature of SAT solving, combined with the need to iteratively solve a number of SAT problems presents an important limitation of the proposed approach to perform completeness and consistency analysis. The use of the UPPAAL tool suite for safety and liveness analysis also creates scalability hurdles. However, these scalability issues are directly related to the use of a model checker and not to the TASM language. While model checkers have improved their scalability through heuristics and through the constant increase in computing power [69], model checking as a technology will always face challenges in terms of scalability. The advent of bounded model checking [126] can aid to remedy this situation and will be considered in future work. The analysis of execution time also suffers from scalability issues on two counts. First, by using an observer automaton in addition to the system model, the addition of a parallel automaton creates multiplicative growth in the resulting system model in terms of locations. Since the observer automaton used in the iterative bounded liveness approach contains only 3 locations, this growth is manageable. Second, the approach to measure execution time of the system model uses a combination of the UPPAAL verifier and the UPPAAL simulator, in an iterative fashion to converge on a WCET and on a BCET. Clearly, for a problem of challenging size where verifying a single iteration would prove expensive, doing so iteratively would prove even more intractable. As mentioned in the discussion in Section 8.4.7, when scalability issues arise, the sophistication of UPPAAL can be leveraged to use the heuristics for state space approximation instead of obtaining exact solutions. Whether an approximate solution is feasible or desirable depends on the nature of the problem and the goal of the analysis.

For the analysis of resource consumption behavior, the algorithm provided in Section 5.4 uses brute-force search to iterate through all combinations of rule guards to converge on the minimum and maximum resource consumptions. Clearly, the performance of this algorithm grows exponentially with the number of main machines in the TASM model. Furthermore, the algorithm also relies on the main machines to be "flattened", which could lead to the exponential growth observed in the production cell case study. However, given the case studies analyzed in Chapter 8 and other related experience with modeling resources, the number of resource annotations throughout the entire model is typically limited. As was the case in the production cell case study, if a machine does not contain annotations, it can be removed from the iteration algorithm, leading to improved performance.

The last facet of the presented framework where scalability could influence the feasibility of the approach is in the generation of test cases. The test case generation approach is afflicted by the same scalability issues related to the translation to *SAT* and the use of *SAT* solvers. However, since the rule coverage criterion is used, the number of necessary test cases in a unit test suite for a given machine will always be equal to the number of rules of the machine. Furthermore, for integration testing and the use of sub machines, the number of required test cases to satisfy the rule coverage criterion should be equal to the largest number of rules of an individual machine in the sub machines used in the hierarchical composition. For the case of regression testing, the worst-case behavior would require that every test case of all test suites be generated anew. In this situation, the number of generated test cases will be linear in the total number of rules in the model. However, each test cases will be generated using the translation to *SAT* and running the instance through the solver.

9.2.6 Overall Limitations

The scalability of the proposed framework, especially in terms of static analysis capabilities, presents limitations to the applicability of the framework to large case studies. Furthermore, the translation to the timed automata of UPPAAL and to the Boolean formulas for *SAT* solvers require that a finite version of the TASM model be computable. This is most easily achieved by removing the use of decimal datatypes in the TASM specification. In Section 9.3.3, other analysis engines are surveyed as candidates for use in the framework. More specifically, the use of Satisfiability Modulo Theory (SMT) solvers [21] could provide a viable solution to remedy the limitations regarding the use of decimal datatypes.

Another limitation of the framework is the lack of a generic translation mechanism to easily incorporate novel analysis engines in the framework, as provided through model transformation [179]. In the current version of the framework, the addition of a new input language for another analysis engine would require that the TASM syntax be parsed into the syntax of the new input language, inside of the toolset. Integrating model transformation concepts into the framework could remedy the need to manually hardcode syntax parsing inside of the toolset. Finally, as explained in the evaluation of the TASM language, the language does not currently include features for specifying continuous behavior. Consequently, the proposed framework cannot model hybrid systems. This limitation could be remedied by integrating an environment for hybrid system modeling and analysis, such as the ones presented in [3] andf in [147].

9.2.7 Lessons Learned

Creating formal models of system behavior requires discipline and intellectual investment. Throughout the model development process in the case studies, different levels of model "correctness" were required. For example, having a model written on paper is fairly straightforward to achieve and one can be easily convinced that the model is correct, using simple arguments. However, when the model is captured through a tool and the model is simulated, various flaws in the model were encountered, including syntax errors, deadlocks, and errors in the control logic. Once these errors were ironed out through simulation scenarios and rational arguments, the model was analyzed for completeness and consistency and through the Uppaal tool suite. During the static analysis process, even more errors were encountered, requiring a deeper understanding of the model. The layers of required discipline was interesting to experience because it validates two basic principles of the presented research – the benefits of modeling and the necessity of tool support. Making the necessary effort to create a model will undoubtedly lead to a deeper understanding of the system being designed. This understanding is independent of the language or the specific design approach and is purely a byproduct of the intellectual investment required for modeling. But, as was experienced throughout the case studies, creating a model is only the first step in gaining understanding of the system behavior. The automated analysis tools certainly provided sanity checks to ensure that assumptions about the system can be easily verified. The automated tool support certainly keeps the modeler honest.

9.3 Opportunities for Future Research

This section provides an overview of extensions that can be made to the presented research. Since the research presents a framework, it is extensible by definition. The extensions are grouped into features of the TASM language, features of the framework, and other types of analysis engines that can be integrated into the framework.

9.3.1 Language Extensions

The TASM language provides a minimal syntax to describe system behavior in the form of an abstract machine. Simple extensions to ease the writing of specifications could include the definition of arrays and data structures, common facilities which have proved useful in programming language [239]. Furthermore, in terms of ASM theory, the TASM language is expressed in "block form" [110]. The TASM language could equivalently be expressed in "free form", introducing the *step* construct to delineate the content of a step, as used in [42]. It would be interesting to investigate usability issues in terms of "block form" versus "free form" even though the two forms are semantically equivalent [110]. Furthermore, in the TASM language, communication between different main machines is achieved via shared variables only. In certain contexts, making the interactions explicit can help understand the dependencies between parallel entities, all the while enabling stronger refinement theories through interaction subsetting and notions of trace equivalence [133, 147]. Common explicit communication mechanisms include synchronization channels, as used in UPPAAL 's timed automata, which are borrowed from Communicating Sequential Processes (CSP) [131] and from the Calculus of Communicating Systems (CCS) [170]. Other languages, such as Timed Input/Ouput Automata (TIOA), make interactions explicit by partitioning externally visible transitions into input and output actions [147]. Regardless of the mechanism selected to express cross-machine communication, it would be interesting to investigate the effect on semantics and on usability.

Finally, the TASM language describes system behavior through discrete step transitions. While this behavior is adequate to describe the behavior of hardware and software components, embedded controllers often act on continuous dynamics described through differential equations. Future investigations could focus on how continuous dynamics could be integrated into the language and/or the framework, for the sake of hybrid system modeling, simulation, and verification. The ability to include continuous dynamics in the TASM language would most likely be better introduced through an appropriate modeling, simulation, and verification environment for hybrid systems, integrated in the framework, such as those provided by CHARON [3] and by TIOA [147].

9.3.2 Framework Features

The presented framework provides an integrated environment for modeling, validation, and verification of embedded real-time systems. The crux of the approach focuses on the design phase of the engineering lifecycle. Other features of the framework could include code generation and traceability of model features down to the implementation level. Ongoing research is currently performed in the Embedded Systems Laboratory (ESL) concerning code generation and extending the traceability approach to support the RavenSpark implementation language. The RavenSpark language is a combination of Spark Ada [20], a safe subset of Ada, and the Ada Ravenscar tasking profile, a safe set of tasking features [233]. The ability to trace model features down to the implementation level could help validate assumptions made in the model by comparing the assumptions embedded in the model, such as time annotations, to the running time of implemented code. The comparison could be achieved via measurements obtained through state-of-the-art WCET analysis tools for implementation code [90]. If a relation between the TASM language and an implementation such as RavenSpark exists, the test case generation approach can also be bridged between the language of the model and the language of the system being tested. The approach to test case generation presented in Chapter 8 could be tailored to a specific system by piecing together the templates that cover specific rules to create test sequences. Such test sequences would take the form of executing a set of rules in sequence. The appropriate sequences of rules would depend on the system being tested, including the properties of initial states and the properties of observable states.

The exploration of alternate designs is an important part of the design activity [56]. The assumption in the presented research is that the functional decomposition and design alternatives have already been evaluated separately and that a design has been agreed upon. Facilities to provide design tradeoffs and to explore the design trade space would certainly complement the design features of the framework. Finally, most system engineering practice begins through requirements engineering [141]. The ability to integrate the presented framework with established requirement elicitation and classification methods could provide end-to-end traceability of requirements from high-level down to implementation [217]. Furthermore, using the traceability approach with the presented test case generation approach, requirement coverage could potentially be achieved [216]. By integrating with requirement engineering practice and an implementation platform, the presented framework could provide a true end-to-end system engineering framework from requirements to code.

9.3.3 Analysis Engines

The UPPAAL tool suite and the SAT4J *SAT* solver were selected as the analysis engines integrated into the presented framework because they represent two mature engines from two popular branches of analysis – model checking and *SAT* solving. While both types of solvers proved appropriate for the type of analysis provided by the framework, the solvers also have important limitations. The first limitation involves the lack of support for decimal numerical values from the Reals domain. The TASM language provides the facilities for modeling and simulating models containing variables of type `float`, but the translation algorithms do not support the analysis of such TASM models because the relevant engines do not support the datatype. While some workarounds are possible, as suggested in Section B.3.1, the selected engines simply do not provide native support. The second limitation involves the scalability of the selected analysis engines. The "state explosion problem" is a well-known limitation of the model checking approach [67] and also applies to *SAT* solving, which relies on systematically exploring a finite state space. However, with increasing computer power and improved state exploration heuristics, model checking can handle problems of increasing complexity [69]. Furthermore, Bounded Model Checking (BMC) provides a an approach to model checking where scalability can be mitigated by controlling the length of paths to be verified [126]. Nevertheless, the need to generate and explore the state space of the model remains the cornerstone of model checking and *SAT* solving and will continue to present scalability challenges [68]. In the following subsections, different classes of analysis engines are explored as potential options to remedy the limitations of the engines currently used in the presented framework.

Linear Programming Solvers

Linear Programming (LP) solvers are a class of constraint solvers with support for solving linear constraints involving a mix of integer and decimal variables [215]. The use of LP solvers in the presented framework has been investigated in [206]. LP solvers provide an advantage over *SAT* solvers because they can handle symbolic constraints and provide native support for decimal variables. However, the constraints need to be linear and the solvers do not provide native support for the disjunction of constraints. Simulating disjunction of constraints is possible through the so-called "big-M" method [218]. While the "big-M" approach can simulate disjunction, the approach scales extremely poorly when multiple disjunctions are present, as is the case for the analysis of completeness and consistency. Furthermore, when the research was conducted, tool support for LP solving was relatively scarce, with the GNU Linear Programming Toolkit (GLPK) being one of the few mature offerings [96].

Satisfiability Modulo Theory Solvers

Satisfiability Modulo Theory (SMT) is a theory for solving constraints that involve Boolean formulas and arithmetic constraints [229]. The solvers that support the automated analysis of constraints, called SMT solvers, extend *SAT* solvers with the ability to reason about arbitrary constraints over the integer domain and the Reals domain. In a sense, SMT solvers combine the benefits of *SAT* solvers and LP solvers. While non-linear constraints are supported by certain solvers, scalability issues arise in the presence of non-linear constraints. SMT solvers have been used as an alternative to traditional model checkers, especially for bounded model checking [14, 97].

At the time when the presented research was conducted, SMT was a relatively new theory and tool support was rather primitive. Nowadays, SMT solvers are gaining popularity and implementations are increasingly more reliable and more scalable [21]. Mature implementations include the MathSAT solver [53] and the Yices solver [142]. Nevertheless, most constraint solving problems are known to be NP-Complete and hence scalability issues will always be present in model checkers, *SAT* solvers, and SMT solvers [232].

Theorem Provers

During the early days of tool-supported formal verification, approaches to verification were polarized into theorem proving and model checking. Theorem proving differs from model checking in that in does not rely on generating a finite state space for the semantics of the model in order to prove properties [137]. Theorem provers use a set of axioms and rules of inference to prove properties of system models [191] syntactically, through symbolic manipulation. Because they do not rely on exhaustive exploration of the state space, theorem provers do not suffer from the same scalability issues as model checkers do. However, theorem provers present different challenges since the analysis typically needs to be guided by the user and is not completely automated [243]. Furthermore, encoding state transition system semantics into a logical theory is not readily achieved [11]. On the usability front, some progress has been made to use the PVS proof system to verify properties of automata models [13, 12]. Popular and mature theorem prover implementations include Coq [83], HOL [106], Isabelle [209], and PVS [207].

9.4 Closing Thoughts

Engineering complex systems is, by definition, a complex endeavor. While the growing sophistication of algorithms and tools, the constant increase in computing power, and the rigor of mathematical theories all provide attractive trends leaning toward automated engineering, the problem of system engineering remains very much a human problem about complexity management. Consequently, the design and development of solutions to address the challenges of engineering complex systems should target the augmentation of the capabilities of the human engineer, not the replacement of the engineer. The anecdotal experience surrounding ASM, combined with the experience with the TASM language and the presented framework has demonstrated that the modeling paradigm of ASM can be readily grasped by someone with minimal programming experience and synchronizes well with established programming practice.

Moreover, there exists a vast body of research in different, seemingly unrelated disciplines such as software engineering, formal methods, and control theory. Many of the research efforts in disparate disciplines cannot be readily aggregated and synthesized to achieve engineering goals. Hopefully, the research presented in this thesis will lead to increased integration of multi-disciplinary approaches to embedded real-time system engineering.

9.5 Segue into the Appendices

The conclusion presented in this chapter marks the end of the narrative of the thesis. The appendices following this chapter can be read in any order and do not follow a linear sequence. Summary descriptions of the content of each appendix is available in Section 1.4.

Appendix A

TASM Language Reference

This appendix explains the concrete constructs of the TASM language as implemented in the TASM toolset. This appendix can be consulted as a supplement to Chapter 4. More specifically, this appendix describes the logical objects that make up the TASM language in the toolset, the rules for constructing names, the list of reserved keywords, the list of operators, and the general typing rules. Furthermore, the context-free grammar of the TASM language, presented in Section A.2, has been used to implement the compiler for the TASM toolset. Semantic implementation topics, such as operator precedence and calling convention, are explained in Section A.3.

A.1 TASM Objects

The concepts described in Chapter 4 are implemented in a suite of logical objects in the TASM language. This section gives the list of logical objects and their properties, as implemented in the TASM toolset. The concrete syntax of how these objects are expressed is described in Section A.2.

A.1.1 Specification

In the logical objects, a *specification* is the overarching concept or object that includes all other logical objects. A *specification* is the complete document that results from capturing a system design in a model expressed in the TASM language.

A.1.2 Project

The *project* is the top level object that contains the high level metadata of the system specification. The project has three attributes, the project *name*, the project *description*, and the *version* of the syntax. The name and description are self-explanatory. The version of the syntax is used to identify older versions of the syntax to preserve backwards compatibility. Other attributes of the project that might be added in the future might include modification times, authors, etc. There is only one project object per specification.

A.1.3 Environment

The *environment* is the object that is used to represent the "outside world". The environment object contains the list of *user-defined types*, which are finite enumerations, the list of *resources*, which are finite quantities, and the list of *variables*, which are the values that affect and are affected by the execution of the various machines in the specification. The environment is a global object that is accessible by all machine instances.

A.1.4 Main Machine Template

In the TASM language, a main machine definition is a template. A machine template is a parameterized version of a machine that needs to be instantiated through a constructor given as part of the template definition. The use of templates enables reuse of specifications and the ability to have multiple versions of a machine definition for a given system design. The concept of a machine template is analogous to the concept of a class in object-oriented programming languages [239]. Main machine templates are instantiated in a Configuration object.

A main machine template contains three attributes – a set of internal variables, a constructor, and a set of rules. The internal variables are typed variables that are visible only inside the machine. The constructor is used to initialize the machine through instantiation and to assign default values to internal variables. The set of rules is a set of guarded commands that govern the machine execution and its effects on the environment. Each rule also specifies the duration of the rule application and the resources consumed during execution of the rule, according to the principles explained in Chapter 4. Additional attributes of a main machine template include a set of monitored variables and a set of controlled variables. The set of monitored variables is the list of environment variables that are used in the guarding conditions of the rules. The set of controlled variables is the list of environment variables that are used in the effect conditions of the rules.

The main machines are the top-level abstract machines that represent a thread of execution. If more than one main machine is present in a system configuration, the resultant specification contains parallelism, also called a *multi-agent* ASM in the Abstract State Machine community [47]. Instantiating multiple main machines is the way to obtain parallel composition of specifications with interleaving semantics.

A.1.5 Function Machine

The idea behind a function machine is a machine with no side-effects that can be used to define abstractions and macros. A function machine is a machine that takes a set of typed inputs and returns a single typed output. A function machine contains a set of input variables and a single output variable. The set of input variables are typed variables that are used to invoke the machine. The output variable is a typed variable that is used to return a value from the machine when it is invoked. A function machine is not allowed to modify the environment and must compute its output solely based on the input values and the values of its monitored variables.

A.1.6 Sub Machine

A sub machine is similar to a main machine except that the sub machine does not execute in its own thread of execution. Instead, a sub machine executes inside of a main machine and shares the thread of execution of the main machine. Sub machines are used to achieve hierarchical composition. A main machine can use more than one sub machine as part of its definition. A sub machine definition contains the same attributes as a main machine except that it does not contain internal variables nor does it contain a constructor.

A.1.7 Configuration

A configuration corresponds to a simulation scenario. A configuration contains a name and a description so that it can be referenced during simulation. A configuration also contains a list of main machine instantiations, defined by invoking the constructors of the main machine templates. Furthermore, the configuration can contain initial values for the environment variables. The initial values specified in a configuration override the initial values of environment variables specified in the environment. Multiple configurations can exist for a given project and a configuration must be selected to perform simulation.

A.2 Syntax

This section describes the concrete syntax of the TASM language, expressed in plain-text format. The plain-text syntax is the format used to read and write specifications using the TASM toolset and it is the input format for the parser and compiler. In the TASM toolset, a system design can be shown across different windows and other user interface components and does not need to be gathered into a single location.

A.2.1 Notational Conventions

The following notational conventions are used in this section and subsequent sections.

- Each abstract type uses the prefix *TASM*
- Constants are enclosed in single quotes (e.g. 'a', '1', etc.), except where set-theoric notation is used

- The formal grammar uses the basic symbols of Backus-Naur Form (BNF) [18] (e.g., {}, [], <>, etc.)

A.2.2 Names

The use of names is crucial in the TASM language; every type of object (variable, type, resource, machine, etc.) is uniquely identified by its name. We define the generic abstract type *TASMName* to express the restrictions on individual names. The type *TASMName* is used in the rest of this document when a name has the listed restrictions. The TASM language has a set of reserved keywords that cannot be used as names. The complete list of reserved keywords is shown in table A.1.

- TASMName is a string of characters
- Each character of TASMName can be either 'a'-'z' or 'A'-'Z' or '_' or '1'-'9' or '-'
- TASMName must start with 'a'-'z' or 'A'-'Z'
- TASMName has a length: 1-64
- TASMName is not a reserved keyword
- TASMName is case-sensitive
- Each TASMName is unique in a given TASM specification

The restrictions on the uniqueness of TASMName's might seem restrictive, especially in the absence of namespaces, but imposing this restriction removes potential ambiguities.

A.2.3 Types

The TASM language contains only simple types. There are no data structures, subtypes, or polymorphic types. The TASM language is also strongly typed; there are no dynamic types or type inference. All typing rules are enforced at compilation time

Table A.1: Reserved keywords

Keyword	Meaning
`t`	Used for time annotations
`next`	Used in time annotations to denote a special value of time
`now`	Used to obtain the value of the global clock
`new`	Used to instantiate a machine template
`Integer`	Denotes the integer datatype
`Float`	Denotes the float datatype
`Boolean`	Denotes the Boolean datatype
`False`	Denotes a constant in the Boolean datatype
`True`	Denotes a constant in the Boolean datatype
`and`	Denotes a logical connective
`or`	Denotes a logical connective
`not`	Denotes a unary operator
`skip`	Denotes the production of an empty update set
`else`	Denotes the special "else rule"
`//`	Used to comment out a given line

and type safety is assured if a TASM specification compiles correctly. The TASM language supplies three default types:

- Integers = $\{\ldots, -1, 0, 1, \ldots\}$
- Floats = *Rational Numbers* (e.g., -1.11, -0.5, 0.0, 10.45, etc.)
- Booleans = $\{True, False\}$

TASM also allows the definition of user-defined types, which are analogous to *enumerations* in most programming languages. However, user-defined types are not assigned integer values and are unordered. A user-defined type is a named type that can be used to provide readable options and type safety. More specifically, a user-defined type is a named type that contains one ore more named values. For example, user-defined types can be defined to denote the status of a light status or the mode of an airplane:

- light_status = {ON, OFF}
- airplane_mode = {Idle, Taxi, Takeoff, Cruise, Landing}

User-defined types are unordered sets of one or more elements where elements must be unique. Each member element is a TASMName. Furthermore, the name of the type is a TASMName.

The TASM language is a strongly typed language, meaning that all variables are typed and that type-safety is enforced at compilation time. No type casting is allowed, even from *Float* to *Integer*. Future versions of the language might allow type casting through functions supplied by the TASM language.

A.2.4 Arithmetic Operators

For *Integer* and *Float* types, the TASM language provides the four basic arithmetic operators, applicable only to operands of the same type:

- addition: $+$

- subtraction: $-$
- multiplication: $*$
- division: $/$

Operations between operands of disparate types is undefined and results in a compilation error. For example, addition between an operand of type *Float* or an operand of type *Integer* results in a compilation error. The arithmetic operators are undefined for Boolean types and for user-defined types.

The assignment operator is the only operator which is defined for all types. Like for the other arithmetic operators, the assignment operator is defined only for operands of the same type:

- assignment: $:=$

The assignment operator does not return a value (denoted by the special character '$\perp$').

A.2.5 Logical Operators

The following two logical operators are defined for all types. The signature of the operators is $Type1 \times Type2 \rightarrow Boolean$ where $Type1 = Type2$. Operators applied to operands of different types are undefined and result in a compilation error.

- equal: $=$
- not equal: $!=$

For *Integer* and *Float* types, the TASM language supplies an additional four logical operators:

- greater than: $>$
- greater than or equal to: $>=$

- less than: <
- less than or equal to: <=

The signature of these operators is also *Type*1 $\times$ *Type*2 $\rightarrow$ *Boolean* where *Type*1 = *Type*2. The logical operators are undefined when the operators are of different types or for *Boolean* and *user-defined* types. For *Boolean* types, the TASM language provides two logical connectives:

- conjunction: *and*
- disjunction: *or*

The signature of these operators is *Boolean* $\times$ *Boolean* $\rightarrow$ *Boolean* and is undefined for non *Boolean* types. For *Boolean* types, the TASM language supplies one unary operator:

- negation: *not*

The signature of this operator is *Boolean* $\rightarrow$ *Boolean* and is undefined for non *Boolean* types.

All operators are summarized in Table A.2.

A.2.6 Context-Free Grammar

The following section explains the formal grammar that is used to express the basic concepts from the previous section, such as types, constants, variables, expressions, etc. The formal grammar is given in Backus-Naur Form (BNF) [18] where the syntactic symbol '|' means "or", '[]' means "optional", and '{}' means 0 or more instances (Kleene closure). The special form of the closure operator, denoted '$\{\}^+$' means 1 or more instances. Any constant is given inside of single quotation marks. For example, the keyword denoting the type integer is given as 'Integer'. It is important to distinguish between the syntactical "optional symbol" ']' and the constant denoting the right bracket "]".

Table A.2: Operators

Operator	Signature	Types
`+`	$Type1 \times Type2 \rightarrow Type3$, $Type1 = Type2 = Type3$	Integer, Float
`-`	$Type1 \times Type2 \rightarrow Type3$, $Type1 = Type2 = Type3$	Integer, Float
`*`	$Type1 \times Type2 \rightarrow Type3$, $Type1 = Type2 = Type3$	Integer, Float
`/`	$Type1 \times Type2 \rightarrow Type3$, $Type1 = Type2 = Type3$	Integer, Float
`:=`	$Type1 \times Type2 \rightarrow \bot$, $Type1 = Type2$	All
`=`	$Type1 \times Type2 \rightarrow Boolean$, $Type1 = Type2$	All
`!=`	$Type1 \times Type2 \rightarrow Boolean$, $Type1 = Type2$	All
`>`	$Type1 \times Type2 \rightarrow Boolean$, $Type1 = Type2$	Integer, Float
`>=`	$Type1 \times Type2 \rightarrow Boolean$, $Type1 = Type2$	Integer, Float
`<`	$Type1 \times Type2 \rightarrow Boolean$, $Type1 = Type2$	Integer, Float
`<=`	$Type1 \times Type2 \rightarrow Boolean$, $Type1 = Type2$	Integer, Float
`and`	$Boolean \times Boolean \rightarrow Boolean$	Boolean
`or`	$Boolean \times Boolean \rightarrow Boolean$	Boolean
`not`	$Boolean \rightarrow Boolean$	Boolean

The BNF grammars describing the concepts of the TASM language is given below. The first part of the grammar supplies the rules for constructing names, constants, and types. The second part of the grammar supplies the rules for constructing expressions, variables, and formulas. The TASM language ignores whitespace unless whitespace is required. When whitespace is required, it is denoted by the token $< TASMWhitespace >$, which represents a single whitespace character. Tab characters, space characters, new line characters, carriage return characters, and form feed characters all represent a single whitespace character.

Basic Concepts

< TASMUCaseLetter >	::=	*'A'* | *'B'* | ... | *'Z'*
< TASMLCaseLetter >	::=	*'a'* | *'b'* | ... | *'z'*
< TASMLetter >	::=	< *TASMUCaseLetter* > | < *TASMLCaseLetter* >
< TASMDigit >	::=	'0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
< TASMCharacter >	::=	< *TASMLetter* > | < *TASMDigit* > | '_'
< TASMASCIIChar >	::=	All standard ASCII characters
< TASMWhiteSpaceChar >	::=	' ' | '\t' | '\n' | '\r' | '\f'
< TASMWhiteSpace >	::=	{< *TASMWhiteSpaceChar* >}$^+$
< TASMIntLit >	::=	['−'] < *TASMDigit* > {< *TASMDigit* >}
< TASMFloatLit >	::=	['−'] < *TASMDigit* > {< *TASMDigit* >} '.' < *TASMDigit* > {< *TASMDigit* >}
< TASMBooleanLit >	::=	*'True'* | *'False'*
< TASMStringLit >	::=	{< *TASMASCIIChar* >}
< TASMName >	::=	< *TASMLetter* > {< *TASMCharacter* >}
< TASMDescription >	::=	< *TASMStringLit* >
< TASMVariable >	::=	< *TASMName* >

```
        < TASMUDTypeName >  ::=  < TASMName >

          < TASMTypeName >  ::=  'Integer' | 'Float' | 'Boolean' | < TASMUDTypeName >

      < TASMUDTypeMember >  ::=  < TASMName >

         < TASMUDTypeDef >  ::=  < TASMUDTypeName >':='
                                 '{'< TASMUDTypeMember > {','< TASMUDTypeMember >}'}'';'

          < TASMConstant >  ::=  < TASMIntLit > | < TASMFloatLit > |
                                 < TASMBooleanLit > | < TASMUDTypeMember >

             < TASMValue >  ::=  < TASMVariable > | < TASMConstant >

       < TASMMachineName >  ::=  < TASMName >

      < TASMFMachineCall >  ::=  < TASMMachineName >'('[< TASMArithExpr > {','< TASMArithExpr >}]')'

         < TASMValueExpr >  ::=  < TASMValue > | < TASMFMachineCall > | 'now'

           < TASMArithOp >  ::=  '+' | '-' | '*' | '/'

         < TASMArithExpr >  ::=  < TASMValueExpr > |
                                 < TASMArithExpr >< TASMArithOp >< TASMArithExpr > |
                                 '('< TASMArithExpr >< TASMArithOp >< TASMArithExpr >')'

        < TASMBinLogicOp >  ::=  '>=' | '>' | '<=' | '<' | '=' | '!='

         < TASMLogicExpr >  ::=  < TASMBooleanLit > |
                                 < TASMArithExpr >< TASMBinLogicOp >< TASMArithExpr > |
                                 '('< TASMArithExpr >< TASMBinLogicOp >< TASMArithExpr >')'

      < TASMLogicBinConn >  ::=  'and' | 'or'

       < TASMLogicUnConn >  ::=  'not'
```

```
< TASMLogicFormula >   ::=  < TASMLogicExpr > |
                            < TASMLogicFormula >< TASMLogicBinConn >< TASMLogicFormula > |
                            < TASMLogicUnConn >< TASMLogicFormula > |
                            '('< TASMLogicExpr >')' |
                            '('< TASMLogicFormula >< TASMLogicBinConn >< TASMLogicFormula >')' |
                            '('< TASMLogicUnConn >< TASMLogicFormula >')'

         < TASMExpr >  ::=  < TASMArithExpr > | < TASMLogicExpr > | < TASMLogicFormula >

      < TASMVarDecl >  ::=  < TASMTypeName >< TASMWhitespace >< TASMVariable >';'

  < TASMVarDeclInit >  ::=  < TASMTypeName >< TASMWhitespace >< TASMVariable >
                            ' :='< TASMConstant >';'

< TASMNameDescPair >   ::=  'NAME :'< TASMName >< TASMWhitespace >
                            'DESC :'< TASMDescription >

      < TASMVarInit >  ::=  < TASMVariable >':='< TASMConstant >';'
```

Environment

```
< TASMResourceName >  ::=  < TASMName >

 < TASMResourceDef >  ::=  < TASMResourceName >':='' ['< TASMIntLit >',' < TASMIntLit >']'';'

 < TASMChannelName >  ::=  < TASMName >

  < TASMChannelDef >  ::=  'Channel' < TASMWhiteSpace >< TASMChannelName >';'

  < TASMEnvironDef >  ::=  'ENVIRONMENT :'< TASMEnvTypeDef >
                           < TASMEnvVarDef >< TASMEnvChannelDef >
                           < TASMEnvResourceDef >
```

$$
\begin{aligned}
<\textbf{TASMEnvTypeDef}> &::= \ 'TYPES:' \ \{<TASMUDTypeDef>\} \\
<\textbf{TASMEnvVarDef}> &::= \ 'VARIABLES:' \ \{<TASMVarDeclInit>\} \\
<\textbf{TASMEnvChannelDef}> &::= \ 'CHANNELS:' \ \{<TASMChannelDef>\} \\
<\textbf{TASMEnvResourceDef}> &::= \ 'RESOURCES:' \ \{<TASMResourceDef>\}
\end{aligned}
$$

Project

$$
<\textbf{TASMProjectDef}> ::= \ 'PROJECT:'<TASMNameDescPair>
$$

Machine Templates

$$
\begin{aligned}
<\textbf{TASMTemplateDef}> &::= \ 'TEMPLATES:'<TASMMTemplatesDef> \\
&\qquad <TASMSTemplatesDef> \\
&\qquad <TASMFTemplatesDef> \\
<\textbf{TASMMTemplatesDef}> &::= \ 'MAIN\ MACHINES:' \ \{<TASMMTemplateDef>\} \\
<\textbf{TASMSTemplatesDef}> &::= \ 'SUB\ MACHINES:' \ \{<TASMSTemplateDef>\} \\
<\textbf{TASMFTemplatesDef}> &::= \ 'FUNCTION\ MACHINES:' \ \{<TASMFTemplateDef>\}
\end{aligned}
$$

Syntax Common to all Machines

```
      < TASMVariableList >  ::=  {< TASMVariable >';' }

          < TASMRuleName >  ::=  < TASMName >

              < TASMRule >  ::=  < TASMRuleName >' {'
                                 [< TASMTimeSpec >]{< TASMResourceSpec >} < TASMRuleDef >'}'

          < TASMTimeSpec >  ::=  't'' :='' ['< TASMIntLit >',' < TASMIntLit >']'';' |
                                 't'' :='< TASMIntLit >';' |
                                 't'' :='' next'';' |
                                 't'' :='' dt'';'

      < TASMResourceSpec >  ::=  < TASMResourceName >':='' ['< TASMIntLit >',' < TASMIntLit >']'';' |
                                 < TASMResourceName >':='< TASMIntLit >';'

           < TASMRuleDef >  ::=  < TASMRuleGuard >< TASMWhiteSpace >< TASMRuleEffect >

         < TASMRuleGuard >  ::=  'if' < TASMLogicFormula >' then'
                                  | 'else' < TASMWhiteSpace >' then'

        < TASMRuleEffect >  ::=  {< TASMEffectExpression >}+

  < TASMEffectExpression >  ::=  < TASMAssignment > | < TASMSubMachineCall > |
                                 < TASMChannelExpr > | 'skip;'

        < TASMAssignment >  ::=  < TASMVariable >':='< TASMArithExpr >';'

    < TASMSubMachineCall >  ::=  < TASMMachineName >' ('')'';'

       < TASMChannelExpr >  ::=  < TASMChannelName >< TASMChannelOpChar >';'

     < TASMChannelOpChar >  ::=  '?' | '!'
```

Main Machine

< TASMMTemplatesDef >	::=	*'MAIN MACHINE :'< TASMNameDescPair >* *< TASMMVars >< TASMConstr >< TASMRules >*
< TASMMVars >	::=	*< TASMContVars >< TASMMonVars >< TASMIntVars >*
< TASMContVars >	::=	*'CONTROLLED VARIABLES :'< TASMVariableList >*
< TASMMonVars >	::=	*'MONITORED VARIABLES :'< TASMVariableList >*
< TASMIntVars >	::=	*'INTERNAL VARIABLES :'* {*< TASMVarDeclInit >*}
< TASMConstr >	::=	*'CONSTRUCTOR :'< TASMMachineName >'* ('[*< TASMParamList >*]')''{' {*< TASMVarInit >*}'}'
< TASMParamList >	::=	*< TASMParam >* {',' *< TASMParamList >*}
< TASMParam >	::=	*< TASMTypeName >< TASMWhiteSpace >< TASMVariable >*
< TASMRules >	::=	*'RULES :'< TASMRule >* {*< TASMWhiteSpace >< TASMRule >*}

Sub Machine

< TASMSTemplateDef >	::=	*'SUB MACHINE :'< TASMNameDescPair >* *< TASMSVars >< TASMRules >*
< TASMSVars >	::=	*< TASMMonVars >< TASMContVars >*

Function Machine

```
< TASMFTemplateDef >  ::=  'FUNCTION MACHINE :'< TASMNameDescPair >
                           < TASMFVars >< TASMRules >

< TASMFVars >  ::=  < TASMInVars >< TASMOutVars >< TASMIntVars >

< TASMInVars >  ::=  'INPUT VARIABLES :' {< TASMVarDecl >}

< TASMOutVars >  ::=  'OUTPUT VARIABLE :'< TASMVarDecl >
```

Configurations

```
< TASMConfigurations >  ::=  'CONFIGURATIONS :' {< TASMConfiguration >}

< TASMConfiguration >  ::=  'CONFIGURATION :'< TASMNameDescPair >
                            < TASMConfMInit >< TASMConfVarInit >

< TASMConfMInit >  ::=  'MACHINE INITIALIZATIONS :' {< TASMMachineInstance >}

< TASMMachineInstance >  ::=  < TASMName >':='' new' < TASMMachineName >
                              '('[< TASMConstant > {',' < TASMConstant >}]')'';'

< TASMConfVarInit >  ::=  'VARIABLE INITIALIZATIONS :'
                          {< TASMVarInit >}
```

A.3 Semantics

Three dominant approaches stand out when expressing programming language semantics - operational semantics, denotational semantics, and axiomatic semantics. Denotational semantics has been used successfully for sequential programs, but the paradigm becomes difficult to work with when concurrency is introduced. Axiomatic semantics has been used on smaller programs, but it is not clear that it works well for larger programs or for languages with numerous concepts. Operational semantics could be used to express the TASM semantics because it has concepts analogous to the TASM language, namely, that of an abstract machine progressing through configurations. Operational semantics has been used extensively to specify language semantics, for both sequential and concurrent programs. However, because the ASM paradigm is close to the operational semantics paradigm, an attempt is made to express the semantics of the TASM language using Abstract State Machines (ASM). The motivation is twofold. First, ASMs have been used to specify the semantics of executable languages, including VHDL, Prolog, and SDL. Second, because the TASM language is built on top of the ASM language, it makes sense to use ASM to express the semantics. In a sense, if the semantics are expressed properly, the TASM language could be viewed as "syntactic sugar" on top of the ASM language.

A.3.1 Operator Precedence

The use of parentheses is strongly encouraged to disambiguate operator precedence for language users. However, the TASM language defines rules for operator precedence when parentheses are not used. The precedence rules are listed in Table A.3.

A.3.2 Calling Convention

In the TASM language, all function ASM calls machine instantiations, and variable references use "call-by-value" semantics. There are no pointers or no references in the TASM language, only distinct variables. Machine instances are all different from one another. When variables are assigned to each other, the value gets copied over to the assigned variable. No variable can be "linked" to the same value, using "pointer-like" semantics.

Table A.3: Operator precedence

Operator	Meaning
$*$	Multiplication
$/$	Division
$+$	Addition
$-$	Subtraction
$>=$	Greater than or equal to
$>$	Greater than
$<=$	Less than or equal to
$<$	Less than
$=$	Equal to
$!=$	Not equal to
and	Logical connective 'AND'
or	Logical connective 'OR'
not	Logical negation 'NOT'
$:=$	Assignment

A.3.3 Types

All variables are strongly typed in the TASM language. There is no type-casting and operators are defined only for operators of the same type. The type checking ensures that all operations are type safe at compile-time. Syntactically, decimal values are interpreted with a required "decimal part", which is a period (".") followed by a digit. This is required even from decimal numbers without a decimal part (e.g., 9.0). "9.0" and "9" are constants of different types, namely the first one is of type "Integer" while the second one is of type "Float". There is no type inference or dynamic typing of any sort as all variables are statically typed and cannot be type casted.

A.3.4 Relation to Abstract State Machines

In this section, the execution semantics of the TASM language are formally expressed using ASM. This is accomplished by using Abstract State Machines (ASM), using the syntax from the Lipari guide [113]. The aim of this section is to express the semantics of the extended language using a "desugaring" into the syntax of the Lipari guide. For the syntax, we follow the notational conventions used in both the Lipari guide and the definition of the formal semantics of SDL [92]. For a detailed list of the ASM syntax used to express formal semantics, the reader is referred to the SDL guide [92], pages 25–27. In Chapter 4, Section 4.4, a translation from TASM to timed ASM is given. The translation given in this section is similar except that the ASM version used in this section uses the Lipari guide syntax, which is closer to the classical definition of ASM.

The key extensions to the TASM language have to do with the addition of time passage and resource consumption. To illustrate time passage, the same conventions as in [72, 92] are adopted and a global dynamic and monotonic increasing function is introduced, called *currentTime*:

- **external** *currentTime*: $\rightarrow REAL$

This function is used inside of machines to query the value of the current time. The function is modified by the environment only and returns a monotonically increasing value greater than 0.0.

A.3.5 Sugaring/Desugaring

The extensions to the TASM language have been introduced as "syntactic sugar" on top of the syntax and semantics of the ASM language as expressed in the Lipari guide [113]. In order to map a TASM specification into an ASM specification, two domains are introduced, namely $DTASM$ and $DASM$ to denote the domains of specifications expressed in the TASM language and the ASM language respectively. A function called *Desug* is also introduced. This function maps a TASM specification into an ASM specification. The "desugaring" function is defined for all individual elements of the TASM language (specifications, variables, types, rules, etc.) and maps the TASM elements into elements of the ASM language.

- $Desug : DTASM \rightarrow DASM$

A.3.6 Resource definitions

A resource definition, $Rdef$, in the environment is desugared into a global shared dynamic function:

- $Desug[[Rdef]] = \textbf{shared } Rdef$

The desugaring of the resource definition is a bit more complex with respect to usage, but the execution semantics of resource usage are detailed in sectionA.3.10.

A.3.7 Type definitions

Type definitions, $Tdef$ get desugared into static finite domains:

- $Desug[[Tdef]] = \textbf{static domain } Tdef$

A.3.8 Variables

Controlled and monitored variables inside of machines get desugared into nullary controlled and dynamic functions, respectively.

A.3.9 Rules

The desugaring of the rules is the most complex desugaring in the TASM language, because this is where time and resource utilization play a role. To illustrate the desugaring of rules, an abstract syntax for a rule definition is defined:

- $Rules = (R_i^+)$
- $R_i = (t_i \; r_i^* \; if \; cond_i \; then \; effect_i)$

In the TASM, the set of rules for a given machine is implicitly mutually exclusive. In the ASM language, the mutual exclusion is explicit. The first desugaring of as set of rules is to generate the explicit mutual exclusion:

- $Desug[[Rules]] = Desug[[((t_0 \; r_0^* \; if \; cond_0 \; then \; effect_0) \; ... \; (t_n \; r_n^* \; if \; cond_n \; then \; effect_n))]] =$

  if $cond_0$ **then** $effect_0$

  else if $cond_1$ **then** $effect_1$

  ...

  else if $cond_n$ **then** $cond_n$

The *else* rule guard from the TASM language would get desugared into a simple **else** rule guard of the ASM language. The time annotations get desugared into an environment variable that affects each machine's execution to simulate "durative" actions. Conceptually, once a rule is triggered, a machine sets a specific variable to the duration of the rule application and will not do anything until the rule duration has elapsed. Once the rule duration has elapsed, the machine will generate the appropriate update set atomically and will be free to execute another rule. Desugaring a time annotation for a rule introduces a new branch if the "if" conditions to denote the time. The concept of a "fresh" variable is introduced to denote a newly generated variable whose name is not previously used. The desugaring introduces two variables, one to keep the time when the rule application will finish executing and one to denote that the machine is "busy" doing work. These two variables are denoted by $tcomplete_{fresh}$ and $mbusy_{fresh}$. The $fresh$ underscore is used to indicate that the variable name is introduced by the desugaring and enforces that it does not clash with existing names. Both of these variables also desugar into controlled dynamic functions:

- *Desug[[*$tcomplete_{fresh}$*]]* = **controlled** *tcomplete* initially *-1*
- *Desug[[*$mbusy_{fresh}$*]]* = **controlled** *mbusy* initially *False*
- *Desug[[Rule]]* = *Desug[[(*$(t_i\ r_i^*\ if\ cond_i\ then\ effect_i)$*)]]* =

```
    if/else if $cond_i \wedge mbusy_{fresh} = False$ then
      $mbusy_{fresh}$ := $True$, $tcomplete_{fresh}$ := $currentTime + getDuration(t_i)$
    else if $currentTime = tcomplete_{fresh} \wedge mbusy_{fresh} = True$ then
      $effect_i$, $mbusy_{fresh}$ := $False$, $timcomplete_{fresh}$ := -1
    ...
```

The function *getDuration* is a macro that is created using the condition and the time annotation of the rule. It returns the duration of the rule. If the time annotation is a single value, it returns that value. Otherwise, if the rule annotation is an interval, it returns a value non-deterministically selected from the interval. Using a macro will enable the desugaring to take into account possible concurrency semantics like WCET and BCET as defined in section 5.3. The introduction of the two auxiliary variables and the time conditions will guarantee that the machine will not produce any update sets and that no other rules will be enabled while the machine is executing a rule. This behavior is exactly the desired behavior to simulate "durative" actions.

Resource annotations get desugared as well, but their usage is a bit different than for the time annotations. Resources are modeled as shared dynamic functions. Their values are set during at the beginning of a rule execution and at the end of a rule execution. Fresh variables are also introduced, for each machine, to denote resource usage:

- *Desug[[Rule]]* = *Desug[[((*t_i r_i^* *if* $cond_i$ *then* $effect_i$*)]]* =

```
if/else if $cond_i \wedge mbusy_{fresh} = False$ then
  $mbusy_{fresh} := True$,
  $tcomplete_{fresh} := currentTime + getDuration(t_i)$,
  $r_{i_{fresh}} := getResourceConsumption(r_i)$
else if $currentTime = tcomplete_{fresh} \wedge mbusy_{fresh} = True$ then
  $effect_i$,
  $mbusy_{fresh} := False$,
  $tcomplete_{fresh} := -1$,
  $r_{i_{fresh}} := 0$
...
```

Function machines are desugared as macros and sub machines are desugared just like main machines and they are "inlined" inside the rule where they are invoked.

A.3.10 Execution Semantics

The desugaring of the TASM language into the ASM language is an easy way to express the formal semantics of the TASM language. In the ASM world, every main machine represents an "*Agent*", member of the **shared domain** *AGENT*. The TASM language also introduces concurrency semantics that are slightly different than for the ASM language. In the TASM language, time is used to synchronize the order of execution between different agents. It is the *currentTime* dynamic function that keeps all of the agents executing in a synchronized order. The time annotations create a partial order between the moves of agents. The *currentTime* function increases monotonically, at a rate that is congruent with the smallest step of a given main machine. For example, if the shortest duration of a rule is 3 time units, for all agents in $AGENT$, then the *currentTime* function will increment each time by 3 time units; this is denoted by this smallest value dt, which corresponds to a **static** function.

The one area that remains to be formally specified is the execution semantics of resources. For each resource that is defined in the environment, an agent is created that is used to sum up all of the resources used by existing agents. These new agents are used to ensure that resource usage falls within the specified bounds.

```
Agent $RESOURCE_i$
 controlled $last_{fresh}$ initially 0
 controlled $totalresource_{i_{fresh}}$ initially 0

  if $currentTime = last_{fresh} + dt$ then
  $totalresource_{i_{fresh}} := sum(r_i)$
 else
  if $totalresource_{i_{fresh}} > resource_{i_{max}}$ then
  $RESOURCE\_EXHAUSTED$
```

The role of the sum macro is to sum up all of the resource annotations from executing agents. The $RESOURCE\_EXHAUSTED$ macro simply halts execution to note that a given resource has been exhausted.

Appendix B

Translating TASM Models to *SAT*

This appendix describes how the constraints in TASM models are translated to a Boolean formula in propositional logic, also called an instance of the Boolean satisfiability problem, or *SAT* for short [232]. The purpose of the mapping is to verify certain properties of TASM models using a *SAT* solver, including completeness and consistency, as explained in Section 5.1. This appendix provides all the details of how different facets of the TASM language map to a Boolean formula.

B.1 Preliminaries

In this appendix, the canonical form of TASM models [110] is used to express the mapping algorithm. As a reminder, a TASM machine is canonical form can be expressed as a set of rules R_i, which are composed of rule guards G_i and effect expressions E_i. For a TASM machine with n rules, the canonical form is expressed as:

$$\begin{aligned}
R_1 &\equiv if\ G_1\ then\ E_1 \\
R_2 &\equiv if\ G_2\ then\ E_2 \\
&\vdots \\
R_n &\equiv if\ G_n\ then\ E_n
\end{aligned}$$

B.2 Translation Algorithm

The goal of the translation algorithm is to map TASM guards to Boolean formulas. A Boolean formula is a set of Boolean variables, denoted b_j, connected with the logical connectives ($\wedge$, $\vee$, $\neg$). More information about *SAT* and Boolean formulas is provided in Section 2.7. The translation from TASM to *SAT* involves mapping the rule guards, G_i, to Boolean propositions, b_j, in Conjunctive Normal Form (CNF). The following sections explain how this translation is performed for the various components of the TASM language.

B.2.1 Function Machines

For a rule guard which contains a function machine call, the function machine call is replaced by the function machine definition, much like "inlining" in programming languages [239]. This substitution will create new rules, in accordance to the procedure used in the proof of Theorem 4.1. The original rule with the function machine call will give rise to n new rules, where n is the number of rules of the function machine. The canonical form of a function machine is given below, where the F prefix denotes a function machine:

$$\begin{aligned}
FR_1 \equiv if\ FG_1\ then\ out\_var &:= out\_val_1; \\
FR_2 \equiv if\ FG_2\ then\ out\_var &:= out\_val_2; \\
&\vdots \\
FR_n \equiv if\ FG_n\ then\ out\_var &:= out\_val_n;
\end{aligned} \tag{1}$$

If the rule where the function machine call is invoked is of the form "*if* G_i *then* E_i", the following n rules will be generated in the machine where the function machine is invoked:

$$R_{i1} \equiv if\ G_i\ \wedge\ FG_1\ then\ E_i$$

$$R_{i2} \equiv if\ G_i\ \wedge\ FG_2\ then\ E_i$$

$$\vdots$$

$$R_{in} \equiv if\ G_i\ \wedge\ FG_n\ then\ E_i$$

Where, in each FG_i, the values of input variables are replaced by the parameters passed to the function call and, in each original guard G_i, the invocation of the function machine is replaced by the *out_val*$_i$ corresponding to the FG_i guard. Once this translation has been performed, the translations to *SAT* described in the following sections can be applied without special handling for function machine calls.

B.2.2 Boolean and User-Defined Datatypes

In the TASM language, user-defined datatypes and Boolean datatypes are simple types that can take values for a finite set. Boolean variables can take one of two values (*True* or *False*). User-defined types can take one of multiple values, as defined by the user. In typical specifications, user-defined types rarely exceed five or six members.

The only operations defined for Boolean and user-defined datatypes are the comparison operators, = and ! =. No other operator is allowed for Boolean and user-defined datatypes. In the translation to *SAT* , the equality operator (=) is assumed to mean a non-negated proposition (e.g., b_1). The operator ! = is translated to mean a negated proposition (e.g., $\neg b_1$). The translation to *SAT* for these datatypes involves 2 steps. The first step is generating the *at least one* clause and the *at most one* clause for each variable of type Boolean or of type user-defined type. The second step involves formulating the property to be verified as a clause in CNF, S, according to the definitions in Section 5.1. The *at least one* clause ensures that the variable must take at least one value from its finite set. This clause is simply the disjunction of equality propositions for each possible value that the variable can take. The *at most one clause* is a clause that ensures that each variable can take at most one value from its finite set.

To illustrate the generation of the *at least one* and *at most one* clauses, the following type is introduced: *type*1 := $\{val_1, val_2, \ldots, val_n\}$. A variable of type Boolean can be viewed as a variable of type *type*1 where n = 2. First, the set of propositions is generated. In *SAT* , a proposition is a single letter with a subscript (e.g., b_i). For a variable named *var* of type *type*1, the following propositions would be generated, where the b_i's represent the *SAT* atomic propositions and the right hand side represents the meaning of the proposition in the TASM context:

$$\begin{aligned} b_1 : var &= val_1 \\ b_2 : var &= val_2 \\ &\vdots \\ b_n : var &= val_n \end{aligned}$$

The *at least one* clause, denoted C_1 for this variable would be:

$$C_1 \equiv b_1 \lor b_2 \ldots \lor b_n$$

The *at least one* clause ensures that at least one of the b_i's must be true for the clause to be true. The *at most one* clause ensures that no two b_i's can be true at the same time. The *at most one clause*, denoted C_2 is the conjunction of multiple clauses:

$$
\begin{aligned}
C_2 \equiv\ & (\neg b_1 \vee \neg b_2 \ldots \vee \neg b_n) && \wedge \\
& (b_1 \vee \neg b_2 \ldots \vee \neg b_n) && \wedge \\
& (\neg b_1 \vee b_2 \ldots \vee \neg b_n) && \wedge \\
& \quad\vdots && \wedge \\
& (\neg b_1 \vee \neg b_2 \ldots \vee b_n)
\end{aligned}
$$

The at most one clause generates $n + 1$ clauses, one for the full negations of the propositions and one for each $n - 1$ negations of propositions. This combination ensures that at most one of the clauses can be true. The conjunction $C_1 \wedge C_2$, which is already in conjunctive normal form, serves to enforce the "exactly one value per variable" constraint, also called *type enforcement.* The rule guards are made up of propositions that already exist in the proposition catalog. For each rule guard in the problem formulation S, for each constraint in the guards, if the constraint is of the form $var = val_i$, its corresponding proposition b_i is looked up in the catalog and substituted in the problem formulation S. If, on the other hand, the constraint is of the form $var\,! = val_i$, the b_i corresponding to $var = val_i$ is looked up in the proposition table and the constraint in the guard is substituted by its negation, $\neg b_i$. Once the substitution is done in the rule guards, the formulated problem S is then converted to Conjunctive Normal Form (CNF) using the well-known algorithm in [232]. The result of this substitution and conversion to CNF yields S with only atomic Boolean propositions. The full SAT problem can then be formed by the conjunction of S, C_1, and C_2:

$$\text{Full } SAT \text{ problem} \equiv S \wedge C_1 \wedge C_2$$

B.2.3 Integer Datatypes

Similarly to Boolean datatypes and user-defined datatypes, integer datatypes take values from a finite set. However, the number of values that integers can take is much larger than for Boolean datatypes and much larger than for typical user-defined types. For example, in the TASM language, integers range from -32,768 to 32,767. While the approach suggested above for Boolean and user-defined types might also work for integer types, the enumeration of all 65,536 possible values would be intractable for a single integer variable. The adopted mapping for integer variables relies on the fact that even though integers are used in TASM specifications, they are used in such a way that they could be replaced by user-defined types. In other words, in TASM specifications, the full range of integers is typically not used.

Nevertheless, integer datatypes are more complex than Boolean and user-defined types because more operations are defined for integer datatypes. These operations are comparison operators and arithmetic operators. The comparison operators are $=$, $!=$, $<$, $<=$, $>$, and $>=$. The arithmetic operators are $+$, $-$, $*$, and $/$. For the suggested translation, constraints on integer variables must be of the form $< var >$ $< com\_pop > < expr >$, where $< var >$ is an integer variable $< comp\_op >$ is a comparison operator and $< expr >$ is an arbitrary arithmetic expression that can contain constants, variable references, function machine calls, and operators. The restriction is that the left hand side of constraints can contain only a variable, with no arithmetic expressions allowed. The translation proposed in this section, deals only with linear constraints whose right hand sides are constants. Arbitrary symbolic right hand sides can be addressed in future research, as explained in section 9.3.

The key idea behind the translation is to convert each integer variable to a user-defined type. This is achieved by collecting all of the constraints on a given integer variable and extracting the intervals that are of interest. These intervals become the members of the user-defined types. Once the integer type has been converted to a user-defined type in this fashion, it can then be converted to a Boolean formula using the approach from Section B.2.2. The algorithm to reduce integer variable to user-defined types consists of 4 steps. For each monitored variable of type integer:

1. Collect all constraints on the variable from S
2. Sort all constraints in ascending order of right-hand sides
3. Create unique intervals for constraints that overlap
4. In S, replace original constraints by disjunction of constraints for modified constraints in overlapping intervals

The translation can be illustrated using an example. Steps 1 and 2 of the algorithm are self-explanatory. Steps 3 and 4 are illustrated using an example which contains the following set of constraints:

$$\begin{aligned} a &>= 2 \\ a &< 3 \\ a &\,!= 5 \\ a &> 5 \\ a &= 9 \end{aligned}$$

The non-overlapping intervals for this set of constraints become the possible values for the integer variable. These intervals are shown below, with the associated Boolean propositions in the SAT problem.

$$\begin{aligned} a &<= 1 : b_1 \\ a &= 2 : b_2 \\ 3 <= a &<= 4 : b_3 \\ a &= 5 : b_4 \\ 6 <= a &<= 8 : b_5 \\ a &= 9 : b_6 \\ a &>= 10 : b_7 \end{aligned}$$

Once an integer variable has been reduced to a user-defined type, the original clauses can be replaced by the Boolean propositions or disjunction of Boolean propositions, depending on the nature of the original constraint.

$$\begin{aligned} a &>= 2 : (b_2 \lor b_3 \lor b_4 \lor b_5 \lor b_6 \lor b_7) \\ a &< 3 : (b_1 \lor b_2) \\ a &!= 5 : (\neg b_4) \\ a &> 5 : (b_5 \lor b_6 \lor b_7) \\ a &= 9 : (b_6) \end{aligned}$$

Once the integer variables have been reduced to user-defined types and the constraints in the problem formulation S have been replaced with the appropriate combination of propositions, the full SAT instance can be created using the *at most one* and the *at least one* clauses, in the same fashion as explained in Section B.2.2. For a specifications where there is significant use of integer constraints, the use of Mixed Integer Programming (MIP) solvers could be better suited for completeness and consistency analysis. This option is investigated in [206] and is addressed in Section 9.3.

B.2.4 Constraints with Symbolic Right-Hand Sides

The translation strategy for integer variables relies on the ability to reduce integer variables to user-defined types. This strategy is straightforward for constraints of the type $< var >< comp\_op >< constant >$ because intervals of interest can be easily identified, as explained in Section B.2.3. However, if the right-hand side of constraints contains arbitrary arithmetic expressions containing a mix of variable references, function machine calls, operations, and constants, the reduction to a user-defined type is not trivial. This case is currently handled by reducing symbolic right-hand sides to a constant by using initial conditions combined with a configuration. In the TASM language, a configuration is a set of initial conditions which "overrides" the initial values defined in the environment. This restriction might seem potentially restrictive, but it is probable that there are more restrictions on the variables than the specification expresses. Furthermore, the reduction of symbolic right-hand sides performs constant propagation where applicable and assigns values to free variables. The ability to handle symbolic right-hand sides is considered as part of future work in Section 9.3.

B.2.5 Complete Translation Algorithm

The basic translation principles have been explained in the previous sections. The complete translation algorithm can now be given, for a single machine:

1. Create problem instance S depending on the property to be checked (e.g., consistency or completeness), as explained in Section 5.1

2. Replace function machine calls with extra rules, as explained in Section B.2.1

3. Replace symbolic right-hand sides with values from the chosen configuration

4. Reduce integer variables to user-defined type variables, as explained in Section B.2.3

5. Iterate through all monitored variables and create *at least one* clauses and *at most one* clauses, as explained in section B.2.2

6. Convert problem formulation S to conjunctive normal form and create the full SAT instance, as explained in Section B.2.2

B.3 Analysis

In this section, the limitations of the translation algorithm, as well as the complexity of the translation algorithm are analyzed. While SAT solvers have been heavily optimized and have been able to address problems of industrial size [175], it is important to understand the scalability of the translation algorithm. The scalability of the translation is crucial to ensure that relevant case studies can be analyzed using the translation to SAT . The preliminary results from the translation algorithm indicate that the performance of the translation algorithm might overshadow the performance of the SAT solver.

B.3.1 Limitations

The translation of TASM constraints to SAT rely on the TASM model being finite or on creating a finite version of the model. When a TASM model contains symbolic right-hand sides, the translation algorithm removes the symbolic values using a selected configuration. Clearly, this feature yields an underapproximation of the model, analogous to the approach explained in [110]. For many cases, as the cases studied in Chapter 8, this approximation is adequate.

If a TASM model contains variables of type float, the translation algorithm is not applied in the TASM toolset and the user is notified that the TASM specification does not meet the requirements to be analyzed using SAT . The reason for this limitation is that float variables are, by definition, infinite. Using other types of solvers such as an LP solver [96] or an SMT solver [97] could address these limitations. However, these types of solvers yield other limitations, as explained in Section 9.3.

B.3.2 Complexity Analysis

While the translation algorithm is fairly straightforward, there are a lot of parameters that will affect the space complexity of the translation. The evident parameters being the number of rules (r), the number of monitored variables (v), the number of function machine calls (fc) (with its internal properties like the number of rules inside the function machine), the number of constraints in the guards (c), and the number of members in user defined types (u). For the translation to SAT , the space complexity is characterized by the number of atomic propositions (pr) and the number of clauses (cl) which are generated by the translation. The same definitions are used from Section 2.7.2 - a *proposition* is an atomic Boolean variable. A *clause* is one disjunction block in CNF form.

Space Complexity

For the case of completeness, the number of rules, r, for a given machine is not the dominant factor. The number of rules will increase the number of clauses linearly ($O(r)$). For the case of consistency, since all pairs of rules are considered, the increase in the number of clauses will be quadratic ($O(r^2)$).

For function machine calls, the growth in complexity will be the generation of extra rules in the procedure to eliminate the function machine call. Every function machine call will give rise to a cross-product in the composition of rules, giving rise to exponential growth in the number of rules for each function machine call ($O(r^{(fc+1)})$). The resultant effect on the number of clauses can be inferred by the previous discussion about the clause growth in terms of number of rules for completeness and consistency.

The effect of the number of variables depends heavily on the type of the variables. For Boolean and user-defined types, the number of variables greatly affects the generation of propositions for the *at least one* and *at most one* clauses. Each variable gives rise to linear growth in the number of clauses for the type enforcement clauses ($O(v)$). Each type enforcement clause will contain a proposition for every user-defined type member, u, giving rise to multiplicative growth for the total number of propositions

$(O(v * u))$.

For integer datatypes, the nature of the constraints on the integer variables determines the growth of the reduction to a user-defined type. The number of constraints will give rise to linear growth in the number of new propositions, that is, the number of members of the user-defined type. The dominant growth in the number of total number of propositions comes from the type enforcement clauses and from the replacement of constraints by the user-defined type propositions. The worst-case scenario occurs when a constraint is replaced by approximately the entire disjunction of user-defined type propositions, as is the case for $a >= 2$ in Section B.2.3. However, this case will occur only if there are numerous constraints and this case will be balanced out by other constraints not needing extra propositions. This is clearly illustrated in Section B.2.3 for the example given. In summary, for integer datatypes, the number of new propositions will vary linearly with the number of constraints $(O(c))$. Extra clauses will vary in the same fashion as for Boolean and user-defined types.

B.3.3 Intractability

It is well-known that the satisfiability problem is NP-Complete [232]. By formulating the completeness and consistency problem as a satisfiability problem, the performance of the verification procedure grows exponentially in the worst-case. For the case where there exists a counterexample, that is, a specification is not complete or not consistent, the average case performance can be acceptable. However, establishing completeness and consistency where there is no counterexample involves establishing that the formula is unsatisfiable. The performance of establishing this property will be highly dependent of the actual problem definition and cannot be analyzed a priori, but will require searching the complete state space.

Given the exponential growth that can result from specific problems, the translation to SAT is inadequate for large sets of rules, large sets of variables and complex rule guards. However, because the relationships between the guards are analyzed for a given machine, the number of rules is expected to be manageable from the specifier's perspective, on the order of less than 10 rules. Furthermore, the complexity of the guards should also be tractable from the specifier's standpoint, so it is realistic to expect moderate complexity of the individual guards. The number of function machine calls could greatly affect the feasibility of the analysis since hierarchical structuring of specifications is a natural mechanism in the TASM language.

THIS PAGE INTENTIONALLY LEFT BLANK

Appendix C

Translating TASM Models to UPPAAL's Timed Automata

This appendix describes how TASM models are translated to UPPAAL 's timed automata. The purpose of the mapping is to verify certain properties of TASM models using the UPPAAL tool suite, such as execution time analysis, as explained in Section 5.3. Furthermore, timed automata models can be analyzed for functional correctness using temporal logic formulas [212] and model checking principles [67] using UPPAAL 's tool suite [157], as explained in Section 5.2. The version of UPPAAL used in this thesis is version 4.0.6, released on March 5th, 2007, and available on the UPPAAL web site (http://www.uppaal.com). This appendix provides all the details of how different facets of the TASM language are mapped to the timed automata formalism of UPPAAL .

C.1 Preliminaries

The timed automaton formalism, also called Alur-Dill automata [5], extends finite state automata with a set of real-valued clocks to denote the passage of time. In a timed automaton, all transitions are instantaneous, but time elapses between transitions. Transition guards can contain predicates over clocks to enforce time passage before a transition is taken. State invariants can be used to enforce an upper bound on the time passage in a state. The timed automata used in UPPAAL extend Alur-Dill automata with Integer variables, Boolean variables, committed and urgent locations, and communication channels [24]. In UPPAAL 's timed automata, *locations* correspond to the *states* of Alur-Dill automata. The states of UPPAAL 's automata are a combination of variable values, clock values, and automata locations. *Urgent* locations are used to denote that time should not elapse in a location. *Committed* locations are used to denote an atomic chain of urgent locations. In this thesis, Section 5.3.3 gives detailed description of the timed automaton formalism. For a more extensive description of the timed automaton formalism, the reader is referred to [5, 7]. For a detailed description of UPPAAL and its associated tool suite, the reader is referred to [24, 157, 211].

In the TASM language, described in Chapter 4, the general form of a timed abstract state machine is a set of rules where each rule, R_i, is of the form:

$$R_i \equiv \langle T_i,\ RR_i,\ G_i,\ E_i \rangle \tag{2}$$

Where:

- T_i is the duration of the rule, which is a closed interval $[a\ ,\ b]$ where $a\ \geq 0$, $b\ \geq 0$, and $a\ \leq b$
- RR_i is a set of resources consumed by the rule execution
- G_i is the rule guard, which is a Boolean predicate
- E_i is the rule effect, which is a series of updates to environment variables

This parameterized version of TASM rules is used to describe the translation to UPPAAL 's timed automata.

C.2 Translation Algorithm

The translation algorithm performs the translation of the environment by translating each datatype and variable in the TASM model to an equivalent datatype and variable in UPPAAL 's language, as explained in Section C.2.1. The algorithm then translates each main machine in the model to a timed automaton of UPPAAL , as explained in Section C.2.2. Before the translation of each main machine is performed, the translation algorithm uses the results of Theorem 4.1 and of Theorem 4.2 and removes the hierarchical composition from each main machine. The complete translation algorithm is provided in Section C.2.3 and a sample translation is given in Section C.3.

C.2.1 Variables and Datatypes

The TASM language contains more datatypes than the timed automata language of UPPAAL . For variable datatypes, UPPAAL contains only a bounded integer datatype, a channel datatype, and a clock datatype. Of these datatypes, only the bounded integer datatype can be used to express TASM types and variables. Nevertheless, the bounded integer datatype is generic enough to be able to express the Boolean datatype, the integer datatype, and the user-defined datatype of the TASM language. The integer datatype of the TASM language is mapped directly to the bounded integer datatype of UPPAAL . The Boolean datatype of TASM is mapped to a bounded integer with range "[0, 1]", where "0" means "False" and "1" means "True". The user-defined datatype of the TASM language maps to an appropriately bounded integer type. For a user-defined type that contains m members, the type is translated to an integer datatype with range "[1, m]", where "1" corresponds to the first member of the type and "m" corresponds to the mth member of the type. If a TASM model contains variables of type float, the translation to UPPAAL 's timed automata cannot be performed, and the user is notified that the given model cannot be analyzed using UPPAAL . In its current version, UPPAAL does not support datatypes from the Real domain. The summary of the datatype translations is given in Table C.2.1.

TASM Type	UPPAAL Type	TASM Example	UPPAAL Example
Integer[a, b]	int[a, b]	`Integer[-5, 5] p;`	`int[-5, 5] p;`
Boolean	int[0, 1]	`Boolean b;`	`int[0, 1] b;`
Switch := {ON, OFF}	int[1, 2]	`Switch s;`	`int[1, 2] s;`

Table C.1: Datatype translations

C.2.2 Machines and Rules

The translation to UPPAAL 's timed automata is achieved by mapping each *main machine* of a TASM specification to a timed automaton. In the TASM language, a *main* machine is a unit of concurrency. TASM models and UPPAAL models differ in the paradigm used to model time. In UPPAAL 's timed automata, time is naturally used to express time passage between transitions. In the TASM language, time is used to denote the duration of actions, equivalent to the duration of a transition. This difference in paradigm is important conceptually, but from an expressiveness perspective, it is irrelevant since both paradigms are equally expressive [30]. For example, durative transitions can be expressed in timed automata using an extra intermediate location which is used solely to elapse time. In the intermediate location, the automaton waits for time to elapse until it can resume making transitions.

To express the paradigm of durative transitions from TASM models, for each generated timed automaton during the translation, a clock c is created as a local variable for the automaton. The clock is used to enforce the *durative* actions of the TASM language. The semantics of durative transitions are such that a rule execution lasts a finite amount of time before the effect of the rule execution is reflected in the environment. For the rule of Equation 2, the corresponding timed automaton is depicted in Figure C-1. The set of consumed resources is omitted from the UPPAAL model, because the resource model used in the TASM language does not affect timing behavior and the semantics of transitions. The behavior to be studied using the UPPAAL tool suite, namely timing behavior and functional properties, does not require the inclusion of resource consumption in the timed automata model.

In Figure C-1, the location *pivot* is the initial location and depicts that the corresponding machine is idle, that is, waiting to execute a rule. In UPPAAL , the *pivot*

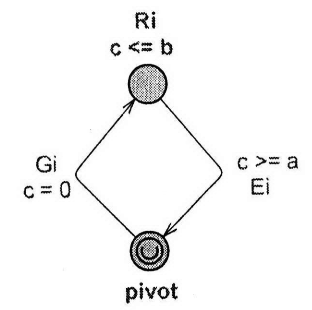


Figure C-1: Timed automaton for rule R_i of Equation 2

location is marked as an *urgent* location, meaning that no time can elapse in this location [157]. Each TASM machine rule R_i will be attached as a transition from the *pivot* location. Because the initial location is urgent, an attached transition will be taken as soon as it is enabled. A transition will be enabled when the rule guard G_i of a rule R_i evaluates to *true*. When this happens, the automaton will transition to a new location called R_i. The name of the location is used to identify the executing rule in order to map the automaton's transitions back to the rule execution of the TASM model. During the transition, the clock c is reset. The location R_i is used to model the durative transitions of the TASM language. The automaton will stay in this location between a and b time units, by using the $c <= b$ location invariant and the $c >= a$ transition guard. Once the duration of the rule has elapsed, the automaton will transition to location *pivot* and the effect of the rule, E_i, will be applied to the environment.

To illustrate the translation of a timed rule, an example is provided. Listing C.1 gives a sample TASM rule. The rule has an execution time ranging between 1 time unit and 4 time units. The corresponding timed automaton is shown in Figure C-2 and contains two locations, one for the *pivot* location, and one for the rule execution.

In the TASM language, a machine can contain a special "*Else rule* "rule, which is a rule that is enabled if no other rule is enabled. The TASM language also contains a special time construct, the " `t := next`" construct. The special combination of both constructs is used to denote that a machine will wait for time to progress until one of its rules becomes enabled. For a machine with n rules, the translation for the

Listing C.1 Sample TASM rule

```
R1: Sample Rule
{
  t := [1, 4];

  if x = y and y = z then
    x := 3;
    y := 5;
}
```

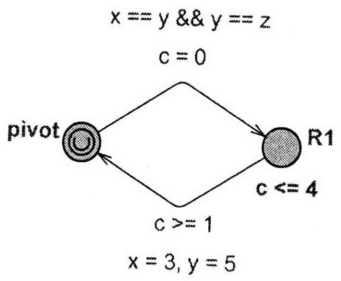


Figure C-2: Timed automaton for Listing C.1

"*Else rule* "with the "`t := next`" annotation is depicted in Figure C-3. The channel *m_else?* is an urgent channel and is used to enforce that the corresponding edge is *urgent* [24]. The urgent edge pattern is used to force the automaton to transition out of location *else* as soon as one its rules becomes enabled. When no rule is enabled, the automaton will stay in location *else* and time will elapse until a rule becomes enabled. When an TASM machine contains the "`t := next`" annotation, another automaton is added to the translation. This automaton contains only one transition and emits a synchronization call along the urgent channel, *m_else!*. This extra automaton is part of the urgent edge pattern [24].

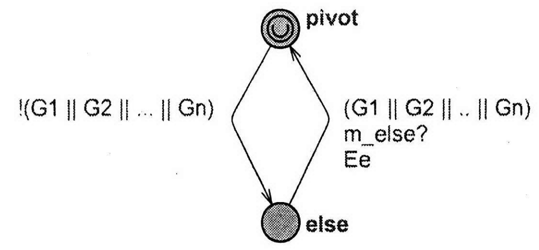


Figure C-3: Timed automaton for the "*Else rule* "and the "`t := next`" annotation

The "`t := next`" annotation and the "*Else rule* "can be used separately from each other. For the rule depicted in Figure C-3, if the "*Else rule* "did not contain the "`t := next`" annotation, the urgent channel would be replaced by a clock *c*, a location invariant, and a clock guard along the edge out of the *else* location, in a fashion identical to the transition depicted in Figure C-1. If, on the other hand, a rule other than the "*Else rule* "contains the "`t := next`" annotation, the translation depicted in Figure C-1 would not have a clock *c*, a location invariant, and an clock predicate in the edge guard. Instead, the transition would contain an urgent channel in the edge guard and the disjunction of all the other guards except for the guard of the rule being executed, similarly to the edge guard out of the *else* transition depicted in Figure C-3.

The translations expressed in Figure C-1 and in Figure C-3 can be generalized to a machine with n rules and an "*Else rule* ". The i subscripts for the clock conditions (e.g., a_1, b_1) denote the durations of the time annotation for the i^{th} rule (e.g., R_1).

For such a machine, the resulting timed automaton would contain $n + 1$ branches, as shown in Figure C-4. In Figure C-4, the "*Else rule* "of the automaton contains the "`t := next`" time annotation.

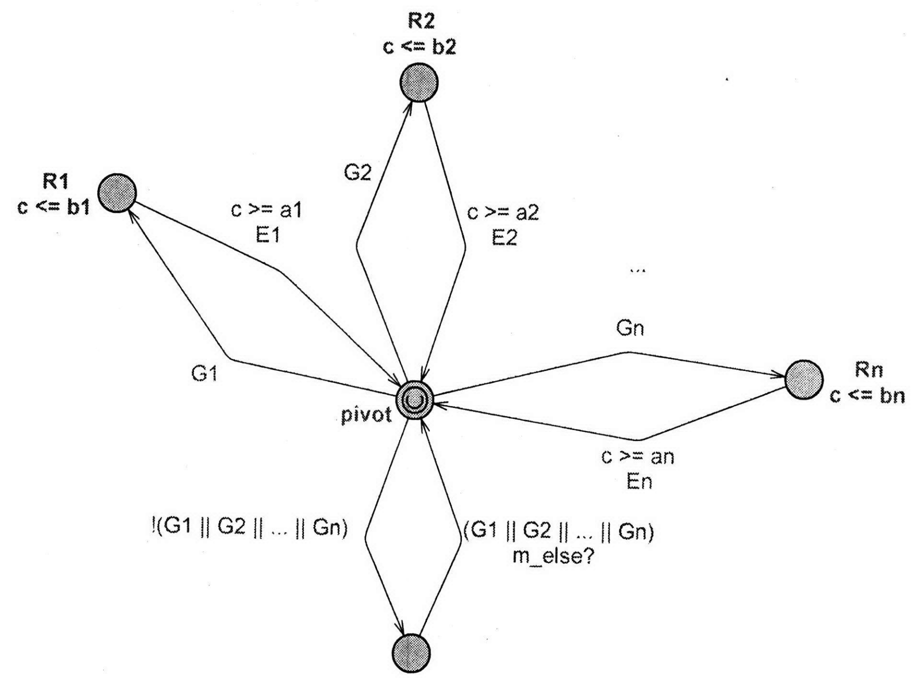


Figure C-4: Timed automaton for a TASM machine with n rules

C.2.3 Complete Translation Algorithm

Given the principles explained in previous sections, the complete translation algorithm can be summarized as follows, for a given TASM model:

1. For each main machine in the TASM model, remove hierarchical composition according to the rules of Theorem 4.1 and of Theorem 4.2
2. Translate the environment:
   (a) Discard resource definitions
   (b) Translate each user-defined type to a corresponding bounded integer type of UPPAAL , as explained in Table C.2.1

(c) Translate each variable and corresponding datatype to the bounded integer type of UPPAAL , as explained in Table C.2.1

3. For each "flattened" main machine

   (a) Create a timed automaton to represent the machine

   (b) Create an initial urgent location called "pivot"

   (c) For each rule R_i of the machine, add a branch from the "pivot" state according to the approach explained in Section C.2.2

   (d) If the machine contains an "else" rule, add an extra branch according to the approach depicted in Section C.2.2

   (e) For rule that contains the "`t := next`" annotation, build an urgent edge using an extra automaton and an urgent channel

C.3 Example

The light switch example from Chapter 4 is used to illustrate the translation from TASM to UPPAAL 's timed automata. In this example, version 4 of the light switch example is used, as depicted in Listing 4.6, in Listing 4.7, and in Listing 4.8. The example does not utilize hierarchical composition so the main machines can be translated directly. The first step of the translation algorithm concerns the translation of the environment. The environment, shown in Listing 4.6, contains two datatypes and four variables. The translation to UPPAAL 's variables is straightforward and is shown below:

```
int[0, 1] light = 1;
int[0, 1] light_switch = 1;
int[0, 1] fan = 1;
int[0, 1] fan_switch = 1;
```

In the translation of the `component_status` datatype, the value `ON` is mapped to the integer "0" and the value `OFF` is mapped to the integer "1". A similar translation is performed for the `switch_status` datatype. The main machine `LIGHT_CONTROL` contains 3 rules and the corresponding timed automaton, shown in Figure C-5, contains 3 branches. Since the machine contains an "*Else rule* "with a "`t := next`" time annotation, the translation also creates an urgent edge, using an urgent channel and an extra timed automaton, shown in Figure C-6. A similar translation is performed for the `FAN_CONTROL` main machine, and the resulting automata are shown in Figure C-7 and in Figure C-8.

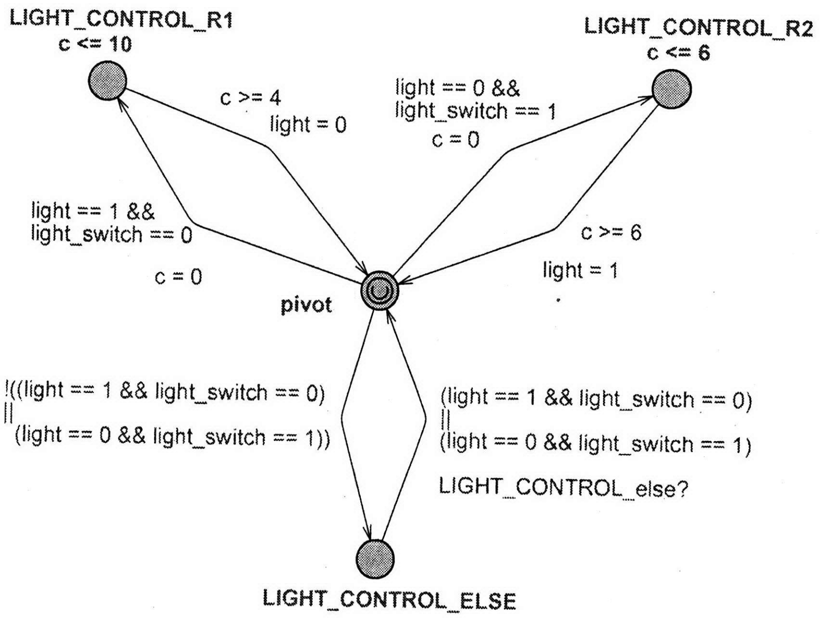


Figure C-5: Timed automaton for the LIGHT_CONTROL machine

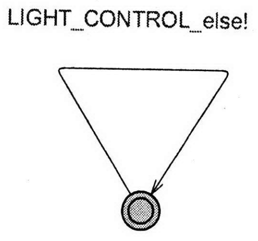


Figure C-6: Timed automaton to enforce the urgent channel for the else rule of machine LIGHT_CONTROL

The complete "Declarations" section of the resulting UPPAAL model is shown in Listing C.2. The complete "Systems declarations" section of the UPPAAL model is shown in Listing C.3. The local "Declarations" sections of the timed automata corre-

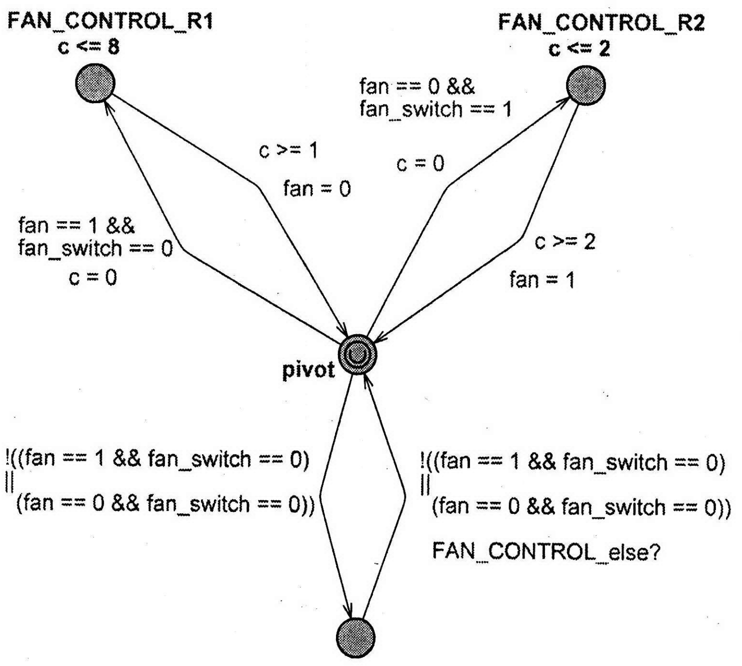


Figure C-7: Timed automaton for the FAN_CONTROL machine

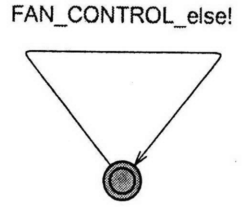


Figure C-8: Timed automaton to enforce the urgent channel for the else rule of machine FAN_CONTROL sponding to each main machine contain only the declaration of the local clock, "`clock c;`", used to enforce the duration of the transitions. The "Declarations" sections of the channel automata do not contain any statements.

Listing C.2 Declarations section of the UPPAAL model

```
int[0, 1] light = 1;
int[0, 1] light_switch = 1;
int[0, 1] fan = 1;
int[0, 1] fan_switch = 1;

urgent chan LIGHT_CONTROL_else;
urgent chan FAN_CONTROL_else;
```

Listing C.3 Systems declarations section of the UPPAAL model

```
light_c         = LIGHT_CONTROL();
light_c_chan    = LIGHT_CONTROL_chan();
fan_c           = FAN_CONTROL();
fan_c_chan      = FAN_CONTROL_chan();

system light_c, light_c_chan, fan_c, fan_c_chan;
```

Appendix D

Production Cell TASM Model

This appendix gives all of the listings for the production cell TASM model. The index of the listings is given in Table D.1. The production cell case study is described in details in [163]. In the context of the TASM language, the case study is described in Section 2.8.1. The TASM model is explained and is analyzed in details in Section 8.1. As a reminder, the logical view of the production cell model is provided in Figure D-1. The complete TASM model contains 8 main machines, one for each component in Figure D-1 and one for the controller. The model also contains 3 function machines, and 16 sub machines.

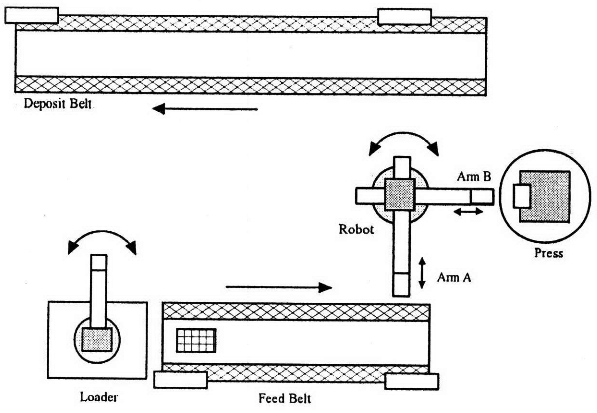


Figure D-1: Top view of the production cell

Name	Type	Purpose	Listing
Types	N/A	List of types	Listing D.1
Variables	N/A	List of variables	Listing D.2, D.3
Loader	Main	Loads blocks onto the feed belt	Listing D.4
Feed	Main	Carries blocks from the loader to the robot	Listing D.5
Deposit	Main	Carries blocks out of the system	Listing D.6
Press	Main	Stamps blocks	Listing D.7
Robot	Main	Simulates the rotation of the robot	Listing D.8
ArmA	Main	Simulates arm a	Listing D.9
ArmB	Main	Simulates arm b	Listing D.10
Controller	Main	Commands the actuators	Listing D.11
armPosition	Function	Returns the position of an arm	Listing D.12
rotateClockwise	Function	Changes the robot angle by $+30^o$	Listing D.13
rotateCounterClockwise	Function	Changes the robot angle by -30^o	Listing D.14
OPERATE_FEED	Sub	Operates the feed belt	Listing D.15
OPERATE_DEPOSIT	Sub	Operates the deposit belt	Listing D.16
OPERATE_ROBOT	Sub	Rotates the robot	Listing D.17
OPERATE_ARM_A	Sub	Operates arm a	Listing D.18
OPERATE_ARM_B	Sub	Operates arm b	Listing D.19
OPERATE_PRESS	Sub	Operates the press	Listing D.20
PICK_UP_ARM_A	Sub	Picks up a block with arm a	Listing D.21
PICK_UP_ARM_B	Sub	Picks up a block with arm b	Listing D.22
DROP_ARM_A	Sub	Drop a block from arm a	Listing D.23
DROP_ARM_B	Sub	Drop a block from arm b	Listing D.24
ARM_A_FEED	Sub	Operates arm a at the feed	Listing D.25
ARM_A_PRESS	Sub	Operates arm a at the press	Listing D.26
ARM_B_DEPOSIT	Sub	Operates arm b at the deposit	Listing D.27
ARM_B_PRESS	Sub	Operates arm b at the press	Listing D.28
ROBOT_MOTION	Sub	Simulates the robot rotation	Listing D.29
ROTATE_ROBOT	Sub	Operates the rotation of the robot	Listing D.30, D.31

Table D.1: List of machines used in the production cell model

D.1 Environment

Listing D.1 User-defined types of the model

```
status          := {empty, loaded};
armposition     := {atfeed, atpress, atdeposit, intransit};
armextension    := {retracted, extended};
Actuator        := {on, off};
Polarity        := {positive, negative};
Stamp           := {notfinished, finished};
Error           := {none, invaliddrop, invalidpickup};
```

Listing D.2 Variables of the model (part 1)

```
//sensors
Integer[0, 90]  robot_angle        := 0;
Stamp           press_block        := notfinished;
Boolean         feed_begin         := False;
armextension    armaext            := retracted;
armextension    armbext            := retracted;
Boolean         feed_end           := False;
Boolean         deposit_begin      := False;
Boolean         deposit_end        := False;

//redundant info, derivable from sensors
armposition     armapos            := atfeed;
armposition     armbpos            := atpress;
status          arma               := empty;
status          armb               := empty;
status          feed_belt          := empty;
status          deposit_belt       := empty;
status          press              := empty;

//other variables
Boolean         wait               := False;
Boolean         robot_wait         := False;
Integer[0, 50]  loaded_blocks      := 0;
Integer[0, 50]  processed_blocks   := 0;
Boolean         loader_done        := False;
Error           error              := none;
armposition     robot_destination  := atfeed;
```

Listing D.3 Variables of the model (part 2)

```
//actuators
Actuator        motor_press        := off;
Actuator        motor_arma         := off;
Actuator        motor_armb         := off;
Actuator        magnet_arma        := off;
Actuator        magnet_armb        := off;
Actuator        motor_robot        := off;
Actuator        motor_feed         := off;
Actuator        motor_deposit      := off;

Polarity        motor_press_p      := positive;
Polarity        motor_arma_p       := positive;
Polarity        motor_armb_p       := positive;
Polarity        motor_robot_p      := positive;
Polarity        motor_feed_p       := positive;
Polarity        motor_deposit_p    := negative;

//constants
Const Integer   ROTATION_ANGLE      := 30;
Const Integer   ARM_B_FEED_ANGLE    := 270;
Const Integer   ARM_B_DEPOSIT_ANGLE := 90;
Const Integer   ARM_B_PRESS_ANGLE   := 0;
Const Integer   ARM_A_FEED_ANGLE    := 0;
Const Integer   ARM_A_PRESS_ANGLE   := 90;
Const Integer   ARM_A_DEPOSIT_ANGLE := 180;
Const Integer   ARM_B_DEPOSIT_ANGLE := 90;
```

D.2 Main Machines

Listing D.4 Rules of the Loader main machine

```
R1: The feed belt is empty, put a block on it {
  t     := 2;
  power := 200;

  if loaded_blocks < number - 1 and feed_belt = empty then
    feed_belt       := loaded;
    loaded_blocks   := loaded_blocks + 1;
    feed_begin      := True;
}

R2: This is the last block... {
  t     := 2;
  power := 200;

  if loaded_blocks = number - 1 and feed_belt = empty then
    feed_belt       := loaded;
    loaded_blocks   := loaded_blocks + 1;
    feed_begin      := True;
    loader_done     := True;
}

R3: The feed belt is loaded, do nothing {
  t     := next;

  if feed_belt = loaded and loaded_blocks < number then
    skip;
}
```

Listing D.5 Rules of the Feed main machine

```
R1: Block goes to end of belt {
  t     := 5;
  power := 500;

  if feed_belt = loaded and feed_begin = True and
     motor_feed = on and motor_feed_p = positive then
    feed_begin  := False;
    feed_end    := True;
}

R2: Else {
  t := next;

  else then
    skip;
}
```

Listing D.6 Rules of the Deposit main machine

```
R1: Block goes to end of belt {
  t     := 7;
  power := 500;

  if deposit_belt = loaded and deposit_begin = True and
     motor_deposit = on and motor_deposit_p = negative then
    deposit_begin   := False;
    deposit_end     := True;
}

R2: Magically take the block out of the system {
    if deposit_end = True then
      deposit_end       := False;
      deposit_belt      := empty;
      processed_blocks  := processed_blocks + 1;
}

R3: Else {
  t := next;

  else then
    skip;
}
```

Listing D.7 Rules of the Press main machine

```
R1: Press is loaded, motor is on {
  t     := 11;
  power := 1500;

  if motor_press = on and press = loaded and press_block = notfinished then
    press_block := finished;
}

R2: Else {
  t := next;

  else then
    skip;
}
```

Listing D.8 Rules of the Robot main machine

```
R1: do {
  if robot_wait = False then
    ROBOT_MOTION();
    robot_wait := True;
}

R2: wait {
  t := next;

  if robot_wait = True then
    robot_wait := False;
}
```

Listing D.9 Rules of the ArmA main machine

```
R1: Extend arm {
  t     := 3;
  power := 1200;

  if motor_arma = on and motor_arma_p = positive and armaext = retracted then
    armaext := extended;
}

R2: Retract arm {
  t     := 2;
  power := 1100;

  if motor_arma = on and motor_arma_p = negative and armaext = extended then
    armaext := retracted;
}

R3: Pick up block {
  if magnet_arma = on and arma = empty then
    PICK_UP_ARM_A();
}

R4: Drop block {
  if magnet_arma = off and arma = loaded then
    DROP_ARM_A();
}

R5: Else {
  t := next;

  else then
    skip;
}
```

Listing D.10 Rules of the ArmB main machine

```
R1: Extend arm {
  t     := 3;
  power := 1200;

  if motor_armb = on and motor_armb_p = positive and armbext = retracted then
    armbext := extended;
}

R2: Retract arm {
  t     := 2;
  power := 1100;

  if motor_armb = on and motor_armb_p = negative and armbext = extended then
    armbext := retracted;
}

R3: Pick up block {
  if magnet_armb = on and armb = empty  then
    PICK_UP_ARM_B();
}

R4: Drop block {
  if magnet_armb = off and armb = loaded then
    DROP_ARM_B();
}

R5: Else {
  t := next;

  else then
    skip;
}
```

Listing D.11 Rules of the Controller main machine

```
R1: Issue Commands {
  if wait = False then
    OPERATE_FEED();
    OPERATE_DEPOSIT();
    OPERATE_ROBOT();
    OPERATE_ARM_A();
    OPERATE_ARM_B();
    OPERATE_PRESS();
    wait := True;
}

R2: Wait for a step {
  t := next;

  else then
    wait := False;
}
```

D.3 Function Machines

Listing D.12 Rules of the armPosition function machine

```
R1: CCW rotation will put arm at feed {
  if value = feed_angle then
    out := atfeed;
}

R2: CCW rotation will put arm at deposit {
  if value = deposit_angle then
    out := atdeposit;
}

R3: CCW rotation will put arm at press {
  if value = press_angle then
    out := atpress;
}

R4: Else, CCW rotation will put arm in transit {
  else then
    out := intransit;
}
```

Listing D.13 Rules of the rotateClockwise function machine

```
R1: Don't go under 0... {
  if robot_angle = 0 then
    out := 360 - ROTATION_ANGLE;
}

R2: Else {
  else then
    out := robot_angle - ROTATION_ANGLE;
}
```

Listing D.14 Rules of the rotateCounterClockwise function machine

```
R1: Don't go over 360... {
  if robot_angle = 360 then
    out := ROTATION_ANGLE;
}

R2: Else {
  else then
    out := robot_angle + ROTATION_ANGLE;
}
```

D.4 Sub Machines

Listing D.15 Rules of the OPERATE_FEED sub machine

```
R1: turn on motor {
  if motor_feed = off and feed_begin = True then
    motor_feed_p := positive;
    motor_feed   := on;
}

R2: turn off motor {
  if motor_feed = on and feed_end = True then
    motor_feed := off;
}

R3: nothing to do {
  else then
    skip;
}
```

Listing D.16 Rules of the OPERATE_DEPOSIT sub machine

```
R1: turn on motor {
  if motor_deposit = off and deposit_begin = True then
    motor_deposit_p     := negative;
    motor_deposit       := on;
}

R2: turn off motor {
  if motor_deposit = on and deposit_end = True then
    motor_deposit := off;
}

R3: nothing to do {
  else then
    skip;
}
```

Listing D.17 Rules of the OPERATE_ROBOT sub machine

```
R1: {
  if armaext = retracted and armbext = retracted then
    ROTATE_ROBOT();
}

R2: Else {
  else then
    skip;
}
```

Listing D.18 Rules of the OPERATE_ARM_A sub machine

```
R1: At feed {
  if armapos = atfeed and motor_robot = off then
    ARM_A_FEED();
}

R2: At press {
  if armapos = atpress and motor_robot = off then
    ARM_A_PRESS();
}

R3: Else {
  else then
    skip;
}
```

Listing D.19 Rules of the OPERATE_ARM_B sub machine

```
R1: At press {
  if armbpos = atpress and motor_robot = off then
    ARM_B_PRESS();
}

R2: At deposit {
  if armbpos = atdeposit and motor_robot = off then
    ARM_B_DEPOSIT();
}

R3: Else {
  else then
    skip;
}
```

Listing D.20 Rules of the OPERATE_PRESS sub machine

```
R1: Turn on press {
  if motor_press = off and press = loaded and press_block = notfinished then
    motor_press := on;
}

R2: Turn off press {
  if motor_press = on and press = loaded and press_block = finished then
    motor_press := off;
}

R3: Else {
  else then
    skip;
}
```

Listing D.21 Rules of the PICK_UP_ARM_A sub machine

```
R1: Pick up at Feed {
  t     := 3;
  power := 1000;

  if armapos = atfeed and armaext = extended and
     arma = empty and feed_end = True then
    feed_end     := False;
    feed_belt    := empty;
    arma         := loaded;
}

R2: Invalid pick up {

  if armapos != atfeed or arma = loaded or
     armaext != extended or feed_end = False then
    error := invalidpickup;
}
```

Listing D.22 Rules of the PICK_UP_ARM_B sub machine

```
R1: Pick up at Press {
  t     := 3;
  power := 1000;

  if armbpos = atpress and armbext = extended and
     armb = empty and press_block = finished then
    press_block := notfinished;
    press       := empty;
    armb        := loaded;
}

R2: Invalid pick up {

  if armbpos != atpress or press_block != finished or
     armb = loaded or armbext != extended then
    error := invalidpickup;
}
```

Listing D.23 Rules of the DROP_ARM_A sub machine

```
R1: Drop at press {
  t     := 2;
  power := 800;

  if armapos = atpress and arma = loaded and
     armaext = extended and press = empty then
    arma            := empty;
    press           := loaded;
    press_block     := notfinished;
}

R2: Invalid drop {
  if armapos != atpress or arma = empty or
     press = loaded or armaext != extended then
    error := invaliddrop;
}
```

Listing D.24 Rules of the DROP_ARM_B sub machine

```
R1: Drop at deposit {
  t     := 2;
  power := 800;

  if armbpos = atdeposit and armb = loaded and
     armbext = extended and deposit_belt = empty then
    armb            := empty;
    deposit_begin   := True;
    deposit_belt    := loaded;
}

R2: Invalid drop {
  if armbpos = intransit or deposit_belt = loaded or armbext != extended then
    error := invaliddrop;
}
```

Listing D.25 Rules of the ARM_A_FEED sub machine

```
R1: extend arm to feed {
  if motor_arma = off and arma = empty and
     armaext = retracted and feed_end = True then
    motor_arma_p    := positive;
    motor_arma      := on;
}

R2: stop motor {
  if motor_arma = on and motor_arma_p = positive and armaext = extended then
    motor_arma := off;
}

R3: pick up block {
  if motor_arma = off and magnet_arma = off and
     arma = empty and armaext = extended then
    magnet_arma := on;
}

R4: retract arm {
  if motor_arma = off and arma = loaded and armaext = extended then
    motor_arma_p    := negative;
    motor_arma      := on;
}

R5: stop motor {
  if motor_arma = on and arma = loaded and armaext = retracted then
    motor_arma := off;
}

R6: Else {
  else then
    skip;
}
```

Listing D.26 Rules of the ARM_A_PRESS sub machine

```
R1: extend arm to press {
  if motor_arma = off and arma = loaded and armaext = retracted then
    motor_arma_p    := positive;
    motor_arma      := on;
}

R2: stop motor {
  if motor_arma = on and arma = loaded and armaext = extended then
    motor_arma := off;
}

R3: drop block in press {
  if motor_arma = off and magnet_arma = on and
     arma = loaded and armaext = extended then
    magnet_arma := off;
}

R4: retract arm {
  if motor_arma = off and arma = empty and armaext = extended then
    motor_arma_p    := negative;
    motor_arma      := on;
}

R5: stop motor {
  if motor_arma = on and arma = empty and armaext = retracted then
    motor_arma := off;
}

R6: Else {
  else then
    skip;
}
```

Listing D.27 Rules of the ARM_B_DEPOSIT sub machine

```
R1: extend arm to deposit {
  if motor_armb = off and armb = loaded and armbext = retracted then
    motor_armb_p    := positive;
    motor_armb      := on;
}

R2: stop motor {
  if motor_armb = on and motor_armb_p = positive and
     armb = loaded and armbext = extended then
    motor_armb := off;
}

R3: drop block {
  if motor_armb = off and magnet_armb = on and
     armb = loaded and armbext = extended then
    magnet_armb := off;
}

R4: retract arm {
  if motor_armb = off and armb = empty and armbext = extended then
    motor_armb_p    := negative;
    motor_armb      := on;
}

R5: stop motor {
  if motor_armb = on and armb = empty and
     motor_armb_p = negative and armbext = retracted then
    motor_armb := off;
}

R6: Else {
  else then
    skip;
}
```

Listing D.28 Rules of the ARM_B_PRESS sub machine

```
R1: extend arm to press {
  if motor_armb = off and armb = empty and
     armbext = retracted and press = loaded then
    motor_armb_p    := positive;
    motor_armb      := on;
}

R2: stop motor {
  if motor_armb = on and motor_armb_p = positive and
     armb = empty and armbext = extended then
    motor_armb := off;
}

R3: pick up block {
  if motor_armb = off and magnet_armb = off and armb = empty and
     armbext = extended and press_block = finished then
    magnet_armb := on;
}

R4: retract arm {
  if motor_armb = off and armb = loaded and armbext = extended then
    motor_armb_p    := negative;
    motor_armb      := on;
}

R5: stop motor {
  if motor_armb = on and motor_armb_p = negative and
     armb = loaded and armbext = retracted then
    motor_armb := off;
}

R6: Else {
  else then
    skip;
}
```

Listing D.29 Rules of the ROBOT_MOTION sub machine

```
R1: rotate clockwise {
  t         := 2;
  power     := 1000;

  if motor_robot = on and motor_robot_p = negative then
    robot_angle := rotateClockwise();
    armapos     := armPosition(ARM_A_FEED_ANGLE, ARM_A_DEPOSIT_ANGLE,
                               ARM_A_PRESS_ANGLE, rotateClockwise());
    armbpos     := armPosition(ARM_B_FEED_ANGLE, ARM_B_DEPOSIT_ANGLE,
                               ARM_B_PRESS_ANGLE, rotateClockwise());
}

R2: rotate counterclockwise {
  t         := 2;
  power     := 1000;

  if motor_robot = on and motor_robot_p = positive then
    robot_angle := rotateCounterClockwise();
    armapos     := armPosition(ARM_A_FEED_ANGLE, ARM_A_DEPOSIT_ANGLE,
                               ARM_A_PRESS_ANGLE, rotateCounterClockwise());
    armbpos     := armPosition(ARM_B_FEED_ANGLE, ARM_B_DEPOSIT_ANGLE,
                               ARM_B_PRESS_ANGLE, rotateCounterClockwise());

}

R3: Else {

    else then
      skip;
}
```

Listing D.30 Rules of the ROTATE_ROBOT sub machine (part 1)

```
R1: all empty, go to feed {
  if motor_robot = off and arma = empty and
     armb = empty and armapos != atfeed then
    motor_robot_p     := negative;
    motor_robot       := on;
    robot_destination := atfeed;
}

R2: all empty, at feed, stop {
  if motor_robot = on and arma = empty and
     armb = empty and armapos = atfeed then
    motor_robot := off;
}

R3: both arms loaded, at feed, go to press {
  if motor_robot = off and arma = loaded and
     armb = loaded and armapos = atfeed then
    motor_robot_p     := positive;
    motor_robot       := on;
    robot_destination := atpress;
}

R4: both arms loaded, at press {
  if motor_robot = on and arma = loaded and
     armb = loaded and armapos = atpress then
    motor_robot := off;
}
```

Listing D.31 Rules of the ROTATE_ROBOT sub machine (part 2)

```
R5: at feed, arm a loaded, press empty, this is the first block {
  if motor_robot = off and arma = loaded and armb = empty and
     press = empty and armapos = atfeed then
    motor_robot_p     := positive;
    motor_robot       := on;
    robot_destination := atpress;
}

R6: at press, arm a loaded, press empty, this is the first block {
  if motor_robot = on and arma = loaded and armb = empty and
     press = empty and armapos = atpress then
    motor_robot     := off;
}

R7: at feed, done loading blocks {
  if motor_robot = off and armapos = atfeed and armb = loaded and
     arma = empty and feed_belt = empty and loader_done = True then
    motor_robot_p     := positive;
    motor_robot       := on;
    robot_destination := atpress;
}

R8: at press, done loading blocks {
  if motor_robot = on and armapos = atpress and arma = empty and
     feed_belt = empty and loader_done = True then
    motor_robot := off;
}

R9: Else {
  else then
    skip;
}
```

THIS PAGE INTENTIONALLY LEFT BLANK

Appendix E

Electronic Throttle Controller TASM Model

This appendix provides the listings for the three TASM models of the Electronic Throttle Controller (ETC) case study. The ETC case study is described in details in [111]. In the context of the TASM language, the case study is described in Section 2.8.2. Section E.1.1 provides the listings for the high level model of the ETC, as explained and analyzed in Section 8.2. Section E.2.1 provides the listings for the tasking and scheduler model of the ETC, as explained and analyzed in Section 8.3. Finally, Section E.3.1 provides the listings for the low level model of the ETC, as explained and analyzed in Section 8.4. Each of the sections provides a brief summary of the model before giving the list of machines used in the model, followed by the actual listings.

E.1 High Level Model

The high level TASM model of the ETC describes the mode switching logic and the logic used to decide on the law for the controller output. The controller output is the desired current, which is calculated based on the controller logic. This version describes a high level model because it does not have any tasks or any calculation of the desired current. The desired current is abstracted using a user-defined type.

The index of the listings is given in Table E.1. The complete TASM model contains 3 main machines, one for the controller, one for the driver behavior, and one for the behavior of the vehicle. The model also contains 13 function machines, and 9 sub machines.

Name	Type	Purpose	Listing
Types	N/A	List of types	Listing E.1
Variables	N/A	List of variables	Listing E.2
CONTROLLER	Main	Performs the controller functions	Listing E.3
DRIVER	Main	Simulates driver behavior	Listing E.4, E.5
VEHICLE	Main	Simulates vehicle behavior	Listing E.6, E.7
Cruise	Function	Determines the cruise control mode	Listing E.8
Cruise_Mode	Function	Sets the cruise mode	Listing E.9
Cruise_Throttle_C	Function	Calculates the cruise mode current	Listing E.10
Driver_Throttle_C	Function	Calculates the human mode current	Listing E.11
Driving_Throttle_C	Function	Calculates the driving mode current	Listing E.12
Fault	Function	Detects if a fault is present	Listing E.13
Limiting_Throttle_C	Function	Calculates the limiting mode current	Listing E.14
Over_Rev	Function	Determines whether the engine revolution is too high	Listing E.15
Over_Rev_Mode	Function	Sets the revolution limiting mode	Listing E.16
Over_Rev_Throttle_C	Function	Calculates the revolution limiting mode current	Listing E.17
Over_Torque	Function	Determines whether the vehicle torque is too high	Listing E.18
Over_Torque_Mode	Function	Sets the traction limiting mode	Listing E.19
Over_Torque_Throttle_C	Function	Calculates the traction limiting mode current	Listing E.20
CALCULATE_OUTPUT	Sub	Wrapper machine to calculate the desired current	Listing E.21
DO_SHUTDOWN	Sub	Performs the shut down functions	Listing E.22
DO_STARTUP	Sub	Performs the start up functions	Listing E.23
HANDLE_FAULT	Sub	Performs the fault tolerance functions	Listing E.24
MONITOR_HEALTH	Sub	Detects the presence of faults	Listing E.25
SAMPLE_STATE	Sub	Reads the state through sensors for the controller	Listing E.26
SET_MAJOR_MODE	Sub	Wrapper machine to set the major controller mode	Listing E.27
SET_MAJOR_MODE_WORK	Sub	Sets the controller major mode	Listing E.28
SET_MINOR_MODE	Sub	Wrapper machine to set the minor controller mode	Listing E.29
SET_MINOR_MODE_WORK	Sub	Sets the controller minor mode	Listing E.30

Table E.1: List of machines used in the high level ETC model

E.1.1 Environment

Listing E.1 User-defined types of the model

```
Binary_Mode     := {active, inactive};
Binary_Status   := {on, off};
Health_Status   := {nominal, fault_detected};
Mode            := {off, startup, shutdown, driving, limiting, faulty};
Gear_Status     := {park, drive};
Control_Mode    := {sample, mode_set_major, mode_set_minor, output, health};
Desired_Current := {none_c, human_c, cruise_c, traction_c, rev_c, min_limiting_c,
                    max_driving_c, fault_c, error_c};
Simulation_Mode := {begin_s, drive_s, random_s, stop_s};
```

Listing E.2 Variables of the model

```
//internal controller modes
Binary_Mode         rev_limiting_mode    := inactive;
Binary_Mode         traction_mode        := inactive;
Binary_Mode         cruise_mode          := inactive;
Mode                controller_mode      := off;
Control_Mode        control_mode         := sample;
Health_Status       system_health        := nominal;
Boolean             startup_done         := False;
Boolean             shutdown_done        := False;

//powertrain sensors
Integer[0, 120]     vehicle_speed        := 0;       //mph
Integer[0, 8000]    engine_speed         := 0;       //rpm
Integer[0, 250]     vehicle_torque       := 0;       //kPa
Boolean             fault                := False;   //is there a fault?

//driver inputs
Binary_Status       ignition             := off;
Binary_Status       cruise_switch        := off;
Integer[0, 45]      pedal_angle          := 0;       //degrees
Gear_Status         gear                 := park;
Binary_Mode         break_pedal          := inactive;

//constants
Const Integer       MAX_ENGINE_SPEED     := 6000;    //rpm
Const Integer       MAX_TORQUE           := 110;     //kPA
Const Integer       MIN_CRUISE_SPEED     := 30;      //mph

//controller inputs
Integer[0, 120]     c_vehicle_speed      := 0;
Integer[0, 8000]    c_engine_speed       := 0;       //rpm
Integer[0, 250]     c_vehicle_torque     := 0;       //kPa
Boolean             c_fault              := False;
Binary_Status       c_ignition           := off;
Binary_Status       c_cruise_switch      := off;
Integer[0, 45]      c_pedal_angle        := 0;       //degrees
Gear_Status         c_gear               := park;
Binary_Mode         c_break_pedal        := inactive;

//controller output
Desired_Current     desired_current      := none_c;

//simulation mode
Simulation_Mode     driver_s             := start_s;
Boolean             vehicle_over_rev_s   := False;
Boolean             vehicle_over_tor_s   := False;
```

E.1.2 Main Machines

Listing E.3 CONTROLLER main machine

```
R1: Controller loop when nominal
{
  if control_mode = sample then
    SAMPLE_STATE();
    control_mode := mode_set;
}

R2: Controller loop to set major mode
{
  if control_mode = mode_set_major then
    SET_MAJOR_MODE();
    control_mode := mode_set_minor;
}

R3: Controller loop to set minor mode
{
  if control_mode = mode_set_minor then
    SET_MINOR_MODE();
    control_mode := output;
}

R4: Controller loop to output current
{
  if control_mode = output then
    CALCULATE_OUTPUT();
    control_mode := health;
}

R5: Controller loop to find failure
{
  if control_mode = health then
    MONITOR_HEALTH();
    control_mode := sample;
}
```

Listing E.4 DRIVER main machine (part 1)

```
R1: Turn on the car
{
  if driver_s = begin_s and controller_mode = off and
     ignition = off then
    ignition := on;
    driver_s := drive_s;
}

R2: Start driving
{
  if driver_s = drive_s and controller_mode = driving and
     vehicle_speed = 0 and gear = park then
    gear          := drive;
    pedal_angle   := 22;
    vehicle_speed := 30;
    driver_s      := random_s;
}

R3: Turn on cruise, slow speed
{
  if driver_s = random_s and cruise_switch = off then
    cruise_switch := on;
    vehicle_speed := 10;
}

R4: Turn on cruise, normal speed
{
  if driver_s = random_s and cruise_switch = off then
    cruise_switch := on;
    vehicle_speed := 30;
}

R5: Turn off cruise
{
  if driver_s = random_s and cruise_switch = on then
    cruise_switch := off;
}

R6: Press break pedal
{
  if driver_s = random_s and break_pedal = inactive then
    break_pedal := active;
}

R7: Depress break pedal
{
  if driver_s = random_s and break_pedal = active then
    break_pedal := inactive;
}
```

Listing E.5 DRIVER main machine (part 2)

```
R8: Stop
{
  if driver_s = random_s then
    gear          := park;
    vehicle_speed := 0;
    ignition      := off;
    driver_s      := stop_s;
}

R9: Do nothing
{
  if driver_s = random_s then
    skip;
}

R10: Stopped
{
  if driver_s = stop_s then
    skip;
}

R11: Else
{
  else then
    skip;
}
```

Listing E.6 VEHICLE main machine (part 1)

```
R1: Randomly change, do nothing
{
  if driver_s = random_s and vehicle_over_rev_s = False and
     vehicle_over_tor_s = False then
    skip;
}

R2: Randomly change RPM
{
  if driver_s = random_s and vehicle_over_rev_s = False and
     vehicle_over_tor_s = False then
    engine_speed       := MAX_ENGINE_SPEED + 1;
    vehicle_over_rev_s := True;
}

R3: Randomly change Traction
{
  if driver_s = random_s and vehicle_over_rev_s = False and
     vehicle_over_tor_s = False then
    vehicle_torque     := MAX_TORQUE + 1;
    vehicle_over_tor_s := True;
}

R4: Randomly change Both
{
  if driver_s = random_s and vehicle_over_rev_s = False and
     vehicle_over_tor_s = False then
    engine_speed       := MAX_ENGINE_SPEED + 1;
    vehicle_over_rev_s := True;
    vehicle_torque     := MAX_TORQUE + 1;
    vehicle_over_tor_s := True;
}

R5: Randomly change RPM, correct
{
  if driver_s = random_s and vehicle_over_rev_s = True and
     vehicle_over_tor_s = False and desired_current = rev_c then
    engine_speed       := MAX_ENGINE_SPEED;
    vehicle_over_rev_s := False;
}
```

Listing E.7 VEHICLE main machine (part 2)

```
R6: Randomly change traction, correct
{
  if driver_s = random_s and vehicle_over_rev_s = False and
     vehicle_over_tor_s = True and desired_current = traction_c then
    vehicle_torque     := MAX_TORQUE;
    vehicle_over_tor_s := False;
}

R7: Randomly change both, correct
{
  if driver_s = random_s and vehicle_over_rev_s = True and
     vehicle_over_tor_s = True and desired_current = min_limiting_c then
    vehicle_torque     := MAX_TORQUE;
    vehicle_over_tor_s := False;
    engine_speed       := MAX_ENGINE_SPEED;
    vehicle_over_rev_s := False;
}

R8: Randomly put in a fault
{
  if driver_s = random_s and fault = False then
    fault := True;
}

R9: Else
{
  else then
    skip;
}
```

E.1.3 Function Machines

Listing E.8 Cruise function machine

```
R1: Cruise condition
{
  if c_vehicle_speed >= MIN_CRUISE_SPEED
     and c_gear = drive
     and c_break_pedal = inactive
     and c_cruise_switch = on then
    outb := True;
}

R2: Else
{
  else then
    outb := False;
}
```

Listing E.9 Cruise_Mode function machine

```
R1: Cruise Active
{
  if Cruise() then
    out := active;
}

R2: Else
{
    else then
      out := inactive;
}
```

Listing E.10 Cruise_Throttle_C function machine

```
R1: Always
{
  if True then
    out := cruise_c;
}
```

Listing E.11 Driver_Throttle_C function machine

```
R1: Always
{
  if True then
    out := human_c;
}
```

Listing E.12 Driving_Throttle_C function machine

```
R1: Cruise enabled, driver input
{
  if cruise_mode = active and c_pedal_angle != 0 then
    out := max_driving_c;
}

R2: Cruise enabled, no driver input
{
  if cruise_mode = active and c_pedal_angle = 0 then
    out := Cruise_Throttle_C();
}

R3: Else
{
  else then
   out := Driver_Throttle_C();
}
```

Listing E.13 Fault function machine

```
R1: Main loop
{
  if fault then
    outb := True;
}

R2: Else
{
  else then
    outb := False;
}
```

Listing E.14 Limiting_Throttle_C function machine

```
R1: Both        //if both over rev and over torque are active
{
  if rev_limiting_mode = active and traction_mode = active then
    out := min_limiting_c;
}

R2: Over_Rev    //if only over rev is active
{
  if rev_limiting_mode = active and traction_mode = inactive then
    out := Over_Rev_Throttle_C();
}

R3: Over_Torque //if only over torque is active
{
  if rev_limiting_mode = inactive and traction_mode = active then
    out := Over_Torque_Throttle_C();
}

R4: Else        //both are inactive. This should never happen!
{
  else then
    out := error_c;
}
```

Listing E.15 Over_Rev function machine

```
R1: Over Rev Condition
{
  if c_engine_speed > MAX_ENGINE_SPEED then
    outb := True;
}

R2: No Over Rev
{
  else then
    outb := False;
}
```

Listing E.16 Over_Rev_Mode function machine

```
R1: Over Rev Mode
{
  if Over_Rev() then
    out := active;
}

R2: Else {
  else then
    out := inactive;
}
```

Listing E.17 Over_Rev_Throttle_C function machine

```
R1: Always
{
  if True then
    out := rev_c;
}
```

Listing E.18 Over_Torque function machine

```
R1: Over Torque Condition
{
  if c_vehicle_torque > MAX_TORQUE then
    outb := True;
}

R2: No Over Torque
{
  else then
    outb := False;
}
```

Listing E.19 Over_Torque_Mode function machine

```
R1: Over Torque Mode
{
  if Over_Torque() then
    out := active;
}

R2: Else
{
  else then
    out := inactive;
}
```

Listing E.20 Over_Torque_Throttle_C function machine

```
R1: Always
{
  if True then
    out := traction_c;
}
```

E.1.4 Sub Machines

Listing E.21 CALCULATE_OUTPUT sub machine

```
R1: Driving Mode
{
  if controller_mode = driving then
    desired_current := Driving_Throttle_C();
}

R2: Limiting Mode
{
  if controller_mode = limiting then
    desired_current := Limiting_Throttle_C();
}

R3: Fault Mode
{
  if controller_mode = faulty then
    HANDLE_FAULT();
}

R4: Startup Mode
{
  if controller_mode = startup then
    DO_STARTUP();
}

R5: Shutdown Mode
{
  if controller_mode = shutdown then
    DO_SHUTDOWN();
}

R6: Fault Mode
{
  if controller_mode = off then
    desired_current := none_c;
}
```

Listing E.22 DO_SHUTDOWN sub machine

```
R1: Do shutdown only when vehicle is stationary
{
  if c_vehicle_speed = 0
     and c_gear = park
     and ignition = off then
    desired_current := none_c;
    shutdown_done   := True;
}

R2: No shutdown
{
  else then
    desired_current := none_c;
    shutdown_done   := False;
}
```

Listing E.23 DO_STARTUP sub machine

```
R1: Do startup only when vehicle is stationary
{
  if c_vehicle_speed = 0 and c_gear = park and
     c_break_pedal = active and c_cruise_switch = off then
    desired_current := none_c;
    startup_done    := True;
}

R2: No startup
{
  else then
    desired_current := none_c;
    startup_done    := False;
}
```

Listing E.24 HANDLE_FAULT sub machine

```
R1: Handle the fault
{
  if c_vehicle_speed = 0 and c_gear = park then
    desired_current := none_c;
    controller_mode := shutdown;
    fault           := False;
    c_fault         := False;
}

R2: Else
{
  else then
    desired_current := fault_c;
}
```

Listing E.25 MONITOR_HEALTH sub machine

```
R1: Find Fault
{
    if Fault() then
      system_health := fault_detected;
}

R2: Else do nothing
{
    else then
      system_health := nominal;
}
```

Listing E.26 SAMPLE_STATE sub machine

```
R1: Cache the state
{
  if True then
    c_vehicle_speed     := vehicle_speed;
    c_engine_speed      := engine_speed;
    c_vehicle_torque    := vehicle_torque;
    c_ignition          := ignition;
    c_cruise_switch     := cruise_switch;
    c_pedal_angle       := pedal_angle;
    c_gear              := gear;
    c_break_pedal       := break_pedal;
}
```

Listing E.27 SET_MAJOR_MODE sub machine

```
R1: No fault
{
  if system_health = nominal then
    SET_MAJOR_MODE_WORK();
}

R2: Else there are faults
{
  if system_health = fault_detected and controller_mode != shutdown then
    controller_mode = faulty;
}

R3: Else
{
  else then
    skip;
}
```

Listing E.28 SET_MAJOR_MODE_WORK sub machine

```
R1: Off -> Startup
{
  if controller_mode = off and ignition = on then
    controller_mode := startup;
}

R2: Startup -> Driving
{
  if controller_mode = startup and startup_done = True then
    controller_mode := driving;
}

R3: Driving -> Limiting
{
  if controller_mode = driving and (Over_Rev() or Over_Torque()) and
     ignition = on then
    controller_mode := limiting;
}

R4: Limiting -> Driving
{
  if controller_mode = limiting and not (Over_Rev() or Over_Torque()) and
     ignition = on then
    controller_mode := driving;
}

R5: Driving, Limiting, Faulty -> Shutdown
{
  if (controller_mode = limiting or controller_mode = driving or
      controller_mode = faulty) and ignition = off then
    controller_mode := shutdown;
}

R6: Shutdown -> Off
{
  if controller_mode = shutdown and shutdown_done = True then
    controller_mode := off;
}

R7: Any other case, do nothing
{
  else then
    skip;
}
```

Listing E.29 SET_MINOR_MODE sub machine

```
R1: No fault
{
  if system_health = nominal then
    SET_MINOR_MODE_WORK();
}

R2: Else
{
  else then
    skip;
}
```

Listing E.30 SET_MINOR_MODE_WORK sub machine

```
R1: Else
{
  if True then
    cruise_mode            := Cruise_Mode();
    rev_limiting_mode      := Over_Rev_Mode();
    traction_mode          := Over_Torque_Mode();
}
```

E.2 Tasking Model

The tasking model describes the tasking structure and scheduler used to implement the electronic throttle controller functionality. The implementation is achieved through 3 tasks, a scheduler, and a 1 ms clock. The index of the listings is given in Table E.2. The complete TASM model contains 3 main machines, 1 function machine, and 10 sub machines.

Name	Type	Purpose	Listing
Types	N/A	List of types	Listing E.31
Variables	N/A	List of variables	Listing E.32
CLOCK	Main	Ticks at 1 ms intervals	Listing E.33
SCHEDULER	Main	Assigns tasks to the processor based on fixed priority and period	Listing E.34
TASKS	Main	Simulates the behavior of the 3 tasks	Listing E.35
finished_to_waiting	Function	Resets completed tasks	Listing E.36
MANAGER_TICK	Sub	Keeps track of manager task period	Listing E.37
MONITOR_TICK	Sub	Keeps track of monitor task period	Listing E.38
SERVO_TICK	Sub	Keeps track of servo task period	Listing E.39
SET_EXECUTING_ TASK	Sub	Assigns execution of a task if the processor is free	Listing E.40
SET_EXECUTION_ PRIORITY	Sub	Decides on the next task to execute based on priority ordering	Listing E.41, E.42
UPDATE_TASK_ STATUSES	Sub	Resets finished tasks to waiting	Listing E.43
WAKE_UP_MANAGER	Sub	Releases manager task on the period	Listing E.44
WAKE_UP_MONITOR	Sub	Releases monitor task on the period	Listing E.45
WAKE_UP_SERVO	Sub	Releases servo task on the period	Listing E.46
WAKE_UP_TASKS	Sub	Wrapper machine to release tasks	Listing E.47

Table E.2: List of machines used in the tasking model of the ETC

E.2.1 Environment

Listing E.31 User-defined types of the model

```
Task_Status := {waiting, released, executing, finished};
Execution   := {done, not_done};
Scheduler   := {wakeup, update, execute, wait};
```

Listing E.32 Variables of the model

```
//Task properties
Task_Status manager_s       := released;
Task_Status monitor_s       := released;
Task_Status servo_s         := released;

//Constants
Const Integer MANAGER_PERIOD := 10;  //in ms
Const Integer MONITOR_PERIOD := 30;  //in ms
Const Integer SERVO_PERIOD   := 3;   //in ms
Const Integer MAJOR_CYCLE    := 30;  //in ms

//Scheduler
Scheduler scheduler_s       := wakeup;
Integer tick                := 0;
Integer oldtick             := 0;
Integer managertick         := 0;
Integer monitortick         := 0;
Integer servotick           := 0;
```

E.2.2 Main Machines

Listing E.33 CLOCK main machine

```
R1: Tick, no reset
{
  t := 1000;

  if tick != MAJOR_CYCLE then
    tick := tick + 1;
    MANAGER_TICK();
    MONITOR_TICK();
    SERVO_TICK();
}

R2: Tick, with reset
{
  t := 1000;

  if tick = MAJOR_CYCLE then
    tick := 1;
    MANAGER_TICK();
    MONITOR_TICK();
    SERVO_TICK();
}
```

Listing E.34 SCHEDULER main machine

```
R1: Step 1, set status
{
  if scheduler_s = update then
    UPDATE_TASK_STATUSES();
    scheduler_s := wakeup;
}

R2: Step 2, wake up tasks
{
  if scheduler_s = wakeup then
    WAKE_UP_TASKS();
    scheduler_s := execute;
}

R3: Step 3, set executing
{
  if scheduler_s = execute then
    SET_EXECUTING_TASK();
    scheduler_s := wait;
}

R4: Wait for a tick
{
  t := 1000;

  if scheduler_s = wait then
    scheduler_s := update;
}
```

Listing E.35 TASKS main machine

```
R1: Execute manager
{
  t := [0, 5];

  if manager_s = executing then
    manager_s := finished;
}

R2: Execute monitor
{
  t := [100, 200];

  if monitor_s = executing then
    monitor_s := finished;
}

R3: Execute servo
{
  t := [70, 100];

  if servo_s = executing then
    servo_s := finished;
}

R4: Else, do nothing, wait for event
{
  t := next;

  else then
    skip;
}
```

E.2.3 Function Machines

Listing E.36 finished_to_waiting function machine

```
R1:
{
  if in = finished then
    out := waiting;
}

R2:
{
  else then
    out := in;
}
```

E.2.4 Sub Machines

Listing E.37 MANAGER_TICK sub machine

```
R1: tick
{
  if managertick = MANAGER_PERIOD then
    managertick := 1;
}

R2: reset
{
  else then
    managertick := managertick + 1;
}
```

Listing E.38 MONITOR_TICK sub machine

```
R1: tick
{
  if monitortick = MONITOR_PERIOD then
    monitortick := 1;
}

R2: reset
{
  else then
    monitortick := monitortick + 1;
}
```

Listing E.39 SERVO_TICK sub machine

```
R1: tick
{
  if servotick = SERVO_PERIOD then
    servotick := 1;
}

R2: reset
{
  else then
    servotick := servotick + 1;
}
```

Listing E.40 SET_EXECUTING_TASK sub machine

```
R1: Someone is still executing, do nothing
{
  if manager_s = executing or monitor_s = executing or
     servo_s = executing then
    skip;
}

R2: Processor is free, assign a task
{
  else then
    SET_EXECUTION_PRIORITY();
}
```

Listing E.41 SET_EXECUTION_PRIORITY sub machine (part 1)

```
R1: All released
{
  if manager_s = released and monitor_s = released and
     servo_s = released then
    manager_s := executing;
}

R2: manager, monitor released
{
  if manager_s = released and monitor_s = released and
     servo_s != released then
    manager_s := executing;
}

R3: manager, servo released
{
  if manager_s = released and servo_s = released and
     monitor_s != released then
    manager_s := executing;
}

R4: monitor, servo released
{
  if monitor_s = released and servo_s = released and
     manager_s != released then
    monitor_s := executing;
}
```

Listing E.42 SET_EXECUTION_PRIORITY sub machine (part 2)

```
R5: only manager
{
  if manager_s = released and monitor_s != released and
     servo_s != released then
    manager_s := executing;
}

R6: only monitor
{
  if monitor_s = released and manager_s != released and
     servo_s != released then
    monitor_s := executing;
}

R7: only servo
{
  if servo_s = released and manager_s != released and
     monitor_s != released then
    servo_s := executing;
}

R8: no one released
{
  if manager_s != released and monitor_s != released and
     servo_s != released then
    skip;
}
```

Listing E.43 UPDATE_TASK_STATUSES sub machine

```
R1: We are at a tick
{
  if tick != oldtick then
    manager_s := finished_to_waiting(manager_s);
    monitor_s := finished_to_waiting(monitor_s);
    servo_s   := finished_to_waiting(servo_s);
}

R2: Not at a tick
{
  else then
    skip;
}
```

Listing E.44 WAKE_UP_MANAGER sub machine

```
R1: wakeup
{
  if manager_s = waiting and managertick = MANAGER_PERIOD then
    manager_s   := released;
}

R2: otherwise
{
  else then
    skip;
}
```

Listing E.45 WAKE_UP_MONITOR sub machine

```
R1: wakeup
{
  if monitor_s = waiting and monitortick = MONITOR_PERIOD then
    monitor_s   := released;
}

R2: otherwise
{
  else then
    skip;
}
```

Listing E.46 WAKE_UP_SERVO sub machine

```
R1: wakeup
{
  if servo_s = waiting and servotick = SERVO_PERIOD then
    servo_s   := released;
}

R2: otherwise
{
  else then
    skip;
}
```

Listing E.47 WAKE_UP_TASKS sub machine

```
R1: wakeup
{
  if oldtick != tick then
    WAKE_UP_MANAGER();
    WAKE_UP_MONITOR();
    WAKE_UP_SERVO();
    oldtick := tick;
}

R2: Else
{
  else then
    skip;
}
```

E.3 Low Level Model

The low level TASM model of the ETC describes the implementation of the mode switching logic and the logic used to decide on the law for the controller output, as implemented through a tasking model. This version combines the functional model from Section E.1 with the tasking model from Section E.2. The index of the listings is given in Table E.3 and in Table E.4. The complete TASM model contains 5 main machines, 14 function machines, and 20 sub machines. In this section, only the modified and new machines are listed. Unchanged machines from previous sections are not repeated and the relevant table entries refer to the listings given in previous sections. The new and changed machines are shown in bold font. The changed machines where only resource consumptions are added are shown in bold and italic font.

Name	**Type**	**Purpose**	**Listing**
Types	N/A	List of types	Listing E.48
Variables	N/A	List of variables	Listing E.50, E.51
Resources	N/A	List of resources	Listing E.49
CLOCK	Main	Ticks at 1 ms intervals	Listing E.33
DRIVER	Main	Simulates the behavior of the driver	Listing E.52, E.53
SCHEDULER	Main	Assigns tasks to the processor based on fixed priority and period	Listing E.54
TASKS	Main	Performs the controller functions	Listing E.55
VEHICLE	Main	Simulates the environment	Listing E.56, E.57
Cruise	Function	Determines the cruise control mode	Listing E.8
Cruise_Mode	Function	Sets the cruise mode	Listing E.9
Cruise_Throttle_C	Function	Calculates the cruise mode current	Listing E.58
Driver_Throttle_C	Function	Calculates the human mode current	Listing E.59
Driving_Throttle_C	Function	Calculates the driving mode current	Listing E.60
Fault	Function	Detects if a fault is present	Listing E.13
Limiting_Throttle_C	Function	Calculates the limiting mode current	Listing E.61
Over_Rev	Function	Determines whether the engine revolution is too high	Listing E.15
Over_Rev_Mode	Function	Sets the revolution limiting mode	Listing E.16
Over_Rev_Throttle_C	Function	Calculates the revolution limiting mode current	Listing E.62
Over_Torque	Function	Determines whether the vehicle torque is too high	Listing E.18

Table E.3: List of machines used in the low level ETC model (part 1)

Name	Type	Purpose	Listing
Over_Torque_Mode	Function	Sets the traction limiting mode	Listing E.19
Over_Torque_Throttle_C	Function	Calculates the traction limiting mode current	Listing E.63
finished_to_waiting	Function	Resets completed tasks	Listing E.36
CALCULATE_OUTPUT	Sub	Wrapper machine to calculate the desired current	Listing E.21
DO_SHUTDOWN	Sub	Performs the shut down functions	Listing E.64
DO_STARTUP	Sub	Performs the start up functions	Listing E.65
HANDLE_FAULT	Sub	Performs the fault tolerance functions	Listing E.66
MANAGER_TICK	Sub	Keeps track of manager task period	Listing E.37
MONITOR_HEALTH	Sub	Detects the presence of faults	Listing E.67
MONITOR_TICK	Sub	Keeps track of monitor task period	Listing E.38
SAMPLE_STATE	Sub	Reads the state through sensors for the controller	Listing E.26
SERVO_TICK	Sub	Keeps track of servo task period	Listing E.39
SET_EXECUTING_ TASK	Sub	Assigns execution of a task if the processor is free	Listing E.40
SET_EXECUTION_ PRIORITY	Sub	Decides on the next task to execute based on priority ordering	Listing E.41, E.42
SET_MAJOR_MODE	Sub	Wrapper machine to set the major controller mode	Listing E.27
SET_MAJOR_MODE_WORK	Sub	Sets the controller major mode	Listing E.28
SET_MINOR_MODE	Sub	Wrapper machine to set the minor controller mode	Listing E.29
SET_MINOR_MODE_WORK	Sub	Sets the controller minor mode	Listing E.30
UPDATE_TASK_ STATUSES	Sub	Resets finished tasks to waiting	Listing E.43
WAKE_UP_MANAGER	Sub	Releases manager task on the period	Listing E.44
WAKE_UP_MONITOR	Sub	Releases monitor task on the period	Listing E.45
WAKE_UP_SERVO	Sub	Releases servo task on the period	Listing E.46
WAKE_UP_TASKS	Sub	Wrapper machine to release tasks	Listing E.47

Table E.4: List of machines used in the low level ETC model (part 2)

E.3.1 Environment

Listing E.48 User-defined types of the model

```
Binary_Mode      := {active, inactive};
Binary_Status    := {on, off};
Health_Status    := {nominal, fault_detected};
Mode             := {off, startup, shutdown, driving, limiting, faulty};
Gear_Status      := {park, drive};
Desired_Current  := {none_c, human_c, cruise_c, traction_c, rev_c, min_limiting_c,
                     max_driving_c, fault_c, error_c};
Simulation_Mode  := {begin_s, drive_s, random_s, stop_s, done_s};
Task_Status      := {waiting, released, executing, finished};
Scheduler        := {wakeup, update, execute, wait, update_state};
Manager_Step     := {major_mode, minor_mode};
```

Listing E.49 Resources of the model

```
memory := [0, 2048000];     //in bytes
power  := [0, 1000000];     //in milliWatts
```

Listing E.50 Variables of the model (part 1)

```
//internal controller modes
Binary_Mode         rev_limiting_mode   := inactive;
Binary_Mode         traction_mode       := inactive;
Binary_Mode         cruise_mode         := inactive;
Mode                controller_mode     := off;
Control_Mode        control_mode        := sample;
Health_Status       system_health       := nominal;
Boolean             startup_done        := False;
Boolean             shutdown_done       := False;

//powertrain sensors
Integer[0, 120]     vehicle_speed       := 0;       //mph
Integer[0, 8000]    engine_speed        := 0;       //rpm
Integer[0, 250]     vehicle_torque      := 0;       //kPa
Boolean             fault               := False;   //is there a fault?

//driver inputs
Binary_Status       ignition            := off;
Binary_Status       cruise_switch       := off;
Integer[0, 45]      pedal_angle         := 0;       //degrees
Gear_Status         gear                := park;
Binary_Mode         break_pedal         := inactive;

//constants
Const Integer       MAX_ENGINE_SPEED    := 6000;    //rpm
Const Integer       MAX_TORQUE          := 110;     //kPA
Const Integer       MIN_CRUISE_SPEED    := 30;      //mph
```

Listing E.51 Variables of the model (part 2)

```
//controller inputs
Integer[0, 120]      c_vehicle_speed      := 0;
Integer[0, 8000]     c_engine_speed       := 0;       //rpm
Integer[0, 250]      c_vehicle_torque     := 0;       //kPa
Boolean              c_fault              := False;
Binary_Status        c_ignition           := off;
Binary_Status        c_cruise_switch      := off;
Integer[0, 45]       c_pedal_angle        := 0;       //degrees
Gear_Status          c_gear               := park;
Binary_Mode          c_break_pedal        := inactive;

//controller output
Desired_Current      desired_current      := none_c;

//simulation mode
Simulation_Mode      driver_s             := start_s;
Boolean              vehicle_over_rev_s   := False;
Boolean              vehicle_over_tor_s   := False;

//Task properties
Task_Status          manager_s            := released;
Task_Status          monitor_s            := released;
Task_Status          servo_s              := released;

//Constants
Const Integer        MANAGER_PERIOD       := 10;  //in ms
Const Integer        MONITOR_PERIOD       := 30;  //in ms
Const Integer        SERVO_PERIOD         := 3;   //in ms
Const Integer        MAJOR_CYCLE          := 30;  //in ms

//Scheduler
Scheduler            scheduler_s          := update_state;
Integer              tick                 := 0;
Integer              oldtick              := 0;
Integer              managertick          := 0;
Integer              monitortick          := 0;
Integer              servotick            := 0;

//Extra variables for refinement
Manager_Step         manager_s_step       := major_mode;
```

E.3.2 Main Machines

Listing E.52 DRIVER main machine (part 1)

```
R1: Turn on the car
{
  if driver_s = begin_s and controller_mode = off and
     ignition = off then
    ignition := on;
    driver_s := drive_s;
}

R2: Start driving
{
  if driver_s = drive_s and controller_mode = driving and
     vehicle_speed = 0 and gear = park then
    gear          := drive;
    pedal_angle   := 22;
    vehicle_speed := 30;
    driver_s      := random_s;
}

R3: Turn on cruise, slow speed
{
  if driver_s = random_s and cruise_switch = off then
    cruise_switch := on;
    vehicle_speed := 10;
}

R4: Turn on cruise, normal speed
{
  if driver_s = random_s and cruise_switch = off then
    cruise_switch := on;
    vehicle_speed := 30;
}

R5: Turn off cruise
{
  if driver_s = random_s and cruise_switch = on then
    cruise_switch := off;
}

R6: Press break pedal
{
  if driver_s = random_s and break_pedal = inactive then
    break_pedal := active;
}
```

Listing E.53 DRIVER main machine (part 2)

```
R7: Depress break pedal
{
  if driver_s = random_s and break_pedal = active then
    break_pedal := inactive;
}

R8: Stop
{
  if driver_s = random_s then
    gear          := park;
    vehicle_speed := 0;
    ignition      := off;
    driver_s      := stop_s;
}

R9: Do nothing
{
  if driver_s = random_s then
    skip;
}

R10: Stopped
{
  if driver_s = stop_s then
    skip;
}

R11: Else
{
  else then
    skip;
}
```

Listing E.54 SCHEDULER main machine

```
R0: Step 0, update state
{
  if scheduler_s = update_state then
    SAMPLE_STATE();
    scheduler_s := update_tasks;
}

R1: Step 1, set status
{
  if scheduler_s = update_tasks then
    UPDATE_TASK_STATUSES();
    scheduler_s := wakeup;
}

R2: Step 2, wake up tasks
{
  if scheduler_s = wakeup then
    WAKE_UP_TASKS();
    scheduler_s := execute;
}

R3: Step 3, set executing
{
  if scheduler_s = execute then
    SET_EXECUTING_TASK();
    scheduler_s := wait;
}

R4: Wait for a tick
{
  t := 1000;

  else then
    scheduler_s := update_state;
}
```

Listing E.55 TASKS main machine

```
R11: Execute manager
{
  t := [0, 3];

  if manager_s = executing and manager_s_step = major_mode then
    SET_MAJOR_MODE();
    manager_s_step := minor_mode;
}

R12: Execute manager
{
  t := [0, 2];

  if manager_s = executing and manager_s_step = minor_mode then
    SET_MINOR_MODE();
    manager_s_step  := major_mode;
    manager_s       := finished;
}

R2: Execute monitor
{
  t := [100, 200];

  if monitor_s = executing then
    MONITOR_HEALTH();
    monitor_s := finished;
}

R3: Execute servo
{
  t := [70, 100];

  if servo_s = executing then
    CALCULATE_OUTPUT();
    servo_s := finished;
}

R4: Else, do nothing, wait for event
{
  t := next;

  else then
    skip;
}
```

Listing E.56 VEHICLE main machine (part 1)

```
R1: Randomly change, do nothing
{
  if driver_s = random_s and vehicle_over_rev_s = False and
     vehicle_over_tor_s = False then
    skip;
}

R2: Randomly change RPM
{
  if driver_s = random_s and vehicle_over_rev_s = False and
     vehicle_over_tor_s = False then
    engine_speed       := MAX_ENGINE_SPEED + 1;
    vehicle_over_rev_s := True;
}

R3: Randomly change Traction
{
  if driver_s = random_s and vehicle_over_rev_s = False and
     vehicle_over_tor_s = False then
    vehicle_torque     := MAX_TORQUE + 1;
    vehicle_over_tor_s := True;
}

R4: Randomly change Both
{
  if driver_s = random_s and vehicle_over_rev_s = False and
     vehicle_over_tor_s = False then
    engine_speed       := MAX_ENGINE_SPEED + 1;
    vehicle_over_rev_s := True;
    vehicle_torque     := MAX_TORQUE + 1;
    vehicle_over_tor_s := True;
}

R5: Randomly change RPM, correct
{
  if driver_s = random_s and vehicle_over_rev_s = True and
     vehicle_over_tor_s = False and desired_current = rev_c then
    engine_speed       := MAX_ENGINE_SPEED;
    vehicle_over_rev_s := False;
}
```

Listing E.57 VEHICLE main machine (part 2)

```
R6: Randomly change traction, correct
{
  if driver_s = random_s and vehicle_over_rev_s = False and
     vehicle_over_tor_s = True and desired_current = traction_c then
    vehicle_torque     := MAX_TORQUE;
    vehicle_over_tor_s := False;
}

R7: Randomly change both, correct
{
  if driver_s = random_s and vehicle_over_rev_s = True and
     vehicle_over_tor_s = True and desired_current = min_limiting_c then
    vehicle_torque     := MAX_TORQUE;
    vehicle_over_tor_s := False;
    engine_speed       := MAX_ENGINE_SPEED;
    vehicle_over_rev_s := False;
}

R8: Randomly put in a fault
{
  if driver_s = random_s and fault = False then
    fault := True;
}

R9: Else
{
  else then
    skip;
}
```

E.3.3 Function Machines

Listing E.58 Cruise_Throttle_C function machine

```
R1: Always
{
  memory := 128;
  power  := 800;

  if True then
    out := cruise_c;
}
```

Listing E.59 Driver_Throttle_C function machine

```
R1: Always
{
  memory := [196, 360];
  power  := [769, 895];

  if True then
    out := human_c;
}
```

Listing E.60 Driving_Throttle_C function machine

```
R1: Cruise enabled, driver input
{
  memory := [324, 826];
  power  := [864, 1695];

  if cruise_mode = active and c_pedal_angle != 0 then
    out := max_driving_c;
}

R2: Cruise enabled, no driver input
{
  if cruise_mode = active and c_pedal_angle = 0 then
    out := Cruise_Throttle_C();
}

R3: Else
{
  else then
   out := Driver_Throttle_C();
}
```

Listing E.61 Limiting_Throttle_C function machine

```
R1: Both        //if both over rev and over torque are active
{
  memory := 648;
  power  := 1425;

  if rev_limiting_mode = active and traction_mode = active then
    out := min_limiting_c;
}

R2: Over_Rev    //if only over rev is active
{
  if rev_limiting_mode = active and traction_mode = inactive then
    out := Over_Rev_Throttle_C();
}

R3: Over_Torque //if only over torque is active
{
  if rev_limiting_mode = inactive and traction_mode = active then
    out := Over_Torque_Throttle_C();
}

R4: Else        //both are inactive. This should never happen!
{
  else then
    out := error_c;
}
```

Listing E.62 Over_Rev_Throttle_C function machine

```
R1: Always
{
  memory := 256;
  power  := 1200;

  if True then
    out := rev_c;
}
```

Listing E.63 Over_Torque_Throttle_C function machine

```
R1: Always
{
  memory := 256;
  power  := 1200;

  if True then
    out := traction_c;
}
```

E.3.4 Sub Machines

Listing E.64 DO_SHUTDOWN sub machine

```
R1: Do shutdown only when vehicle is stationary
{
  memory := 256;
  power  := 900;

  if c_vehicle_speed = 0
     and c_gear = park
     and ignition = off then
    desired_current := none_c;
    shutdown_done   := True;
}

R2: No shutdown
{
  else then
    desired_current := none_c;
    shutdown_done   := False;
}
```

Listing E.65 DO_STARTUP sub machine

```
R1: Do startup only when vehicle is stationary
{
  memory := 128;
  power  := 900;

  if c_vehicle_speed = 0 and c_gear = park and
     c_break_pedal = active and c_cruise_switch = off then
    desired_current := none_c;
    startup_done    := True;
}

R2: No startup
{
  else then
    desired_current := none_c;
    startup_done    := False;
}
```

Listing E.66 HANDLE_FAULT sub machine

```
R1: Handle the fault
{
  memory := 512;
  power  := 895;

  if c_vehicle_speed = 0 and c_gear = park then
    desired_current := none_c;
    controller_mode := shutdown;
    fault           := False;
    c_fault         := False;
}

R2: Else
{
  memory := 512;
  power  := 895;

  else then
    desired_current := fault_c;
}
```

Listing E.67 MONITOR_HEALTH sub machine

```
R1: Find Fault
{
  memory := [512, 1024];
  power  := [1530, 1624];

  if Fault() then
    system_health := fault_detected;
}

R2: Else do nothing
{
  memory := [512, 1024];
  power  := [1530, 1624];

  else then
    system_health := nominal;
}
```

Appendix F

Timeliner Plant Control System TASM Model

This appendix gives all of the listings for the Timeliner plant control system TASM model. The plant control system case study is described in details in [238]. In the context of the TASM language, the case study is described in Section 2.8.3. The TASM models are explained and are analyzed in details in Section 8.5. The complete TASM model contains 5 main machines, no function machines, and 7 sub machines. The machines are listed in Table F.1.

Name	Type	Purpose	Listing
Types	N/A	List of types	Listing F.1
Variables	N/A	List of variables	Listing F.2
Control_Task	Main	Shares the processor with Timeliner	Listing F.3
Scheduler	Main	Switches processor between the control task and Timeliner	Listing F.4
Timeliner	Main	Executes plant control sequences	Listing F.5
Temperature	Main	Simulates the behavior of the cabin temperature	Listing F.6
Humidity	Main	Simulates the behavior of the cabin humidity	Listing F.7
EXECUTE_BUNDLES	Sub	Executes all active bundles	Listing F.8
PLANTSIM_BUNDLE	Sub	Execute the plantsim bundle	Listing F.9
EXECUTE_PLANTSIM _SEQUENCES	Sub	Execute the sequences in the plantsim bundle	Listing F.10
SEQUENCE_TEMP _MONITOR	Sub	Execute the temperature monitor sequence	Listing F.11
SEQUENCE_TEMP _MONITOR_WORK	Sub	Maintains the temperature between 19 and 26 Celsius degrees	Listing F.12, F.13
SEQUENCE_HUMIDITY _MONITOR	Sub	Execute the humidity monitor sequence	Listing F.14
SEQUENCE_HUMIDITY _MONITOR_WORK	Sub	Maintains the humidity between 40 and 60 percent	Listing F.15, F.16

Table F.1: List of machines used in the Timeliner plant control system model

F.1 Environment

Listing F.1 User-defined types of the model

```
Status                  := {active, inactive};
Device                  := {on, off};
Temp_Sequence_Block     := {b0, b1, b2, b3, b4};
Humid_Sequence_Block    := {c0, c1, c2, c3, c4, c5);
Processor_Status        := {timeliner, controltask};
Execution_Status        := {done, not_done};
Bundle                  := {plantsim};
Sequence                := {temp_monitor, humid_monitor};
```

Listing F.2 Variables of the model

```
//Bundle Status
Status                  plantsim_bundle_status      := active;

//Sequence Statuses
Status                  temp_seq_status             := active;
Status                  humid_seq_status            := active;

//Sequence Blocks
Temp_Sequence_Block     temp_seq_b                  := b0;
Humid_Sequence_Block    humid_seq_b                 := c0;

//Bundle Execution Status
Execution_Status        plantsim_s                  := not_done;

//Sequence Execution Statuses
Execution_Status        temp_seq_s                  := not_done;
Execution_Status        humid_seq_s                 := not_done;

//Sequences
Bundle                  exec_bundle                 := plantsim;
Sequence                exec_seq                    := temp_monitor;

//Plant Variables
Boolean                 trying_to_cool_system       := False;
Integer                 temperature                 := 24;
Integer                 humidity                    := 50;

//Devices
Device                  cooling                     := off;
Device                  heating                     := off;
Device                  humidifier                  := off;

//Control Variables
Processor_Status        processor                   := controltask;
Execution_Status        execution                   := not_done;

//Fault control
Boolean                 fault                       := False;
```

F.2 Main Machines

Listing F.3 Rules of the Control_Task main machine

```
R1: Control Task
{
  t := [3500, 5000];

  if processor = controltask and execution = not_done then
    execution := done;
}

R2: Else
{
  t := next;

  else then
    skip;
}
```

Listing F.4 Rules of the Scheduler main machine

```
R1: Controller
{
  t := 1000;

  if processor = controltask and execution = done then
    processor := timeliner;
    execution := not_done;
}

R2: Timeliner
{
  t := 1000;

  if processor = timeliner and execution = done then
    processor := controltask;
    execution := not_done;
}

R3: Else
{
  t := next;

  else then
    skip;
}
```

Listing F.5 Rules of the Timeliner main machine

```
R1: Execute bundles
{

  if processor = timeliner and execution = not_done then
    EXECUTE_BUNDLES();
}

R2: Else
{
  t := next;

  else then
    skip;
}
```

Listing F.6 Rules of the Temperature main machine

```
R1:
{
  t := 685;

  if temperature > 19 and temperature < 26 and
     cooling = off and heating = off and humidifier = off then
    skip;
}

R2:
{
  t := 685;

  if temperature > 19 and temperature < 26 and
     cooling = off and heating = off and humidifier = off then
    temperature     := 18;
}

R3:
{
  t := 685;

  if temperature > 19 and temperature < 26 and
     cooling = off and heating = off and humidifier = off then
    temperature     := 27;
}

R4: Cooling on
{
  t := [0, 1500];

  if temperature > 25 cooling = on and heating = off then
    temperature := 24;
}

R5: Heating on
{
  t := [0, 1500];

  if temperature < 20 and heating = on and cooling = off then
    temperature := 24;
}

R6: Else
{
  t := next;

  else then
    skip;
}
```

Listing F.7 Rules of the Humidity main machine

```
R1:
{
  t := 685;

  if humidity > 39 and humidity < 61 and
     cooling = off and heating = off and humidifier = off then
    skip;
}

R2:
{
  t := 685;

  if humidity > 39 and humidity < 61 and
     cooling = off and heating = off and humidifier = off then
    temperature     := 35;
}

R3:
{
  t := 685;

  if humidity > 39 and humidity < 61 and
     cooling = off and heating = off and humidifier = off then
    temperature     := 65;
}

R4: Cooling on
{
  t := [0, 1500];

  if humidity > 60 and cooling = on and humidifier = off then
    humidity := 50;
}

R5: Humidifier on
{
  t := [0, 1500];

  if humidity < 40 and humidifier = on and cooling = off then
    humidity := 50;
}

R6: Else
{
  t := next;

  else then
    skip;
}
```

F.3 Function Machines

The plant control system does not contain function machines

F.4 Sub Machines

Listing F.8 Rules of the EXECUTE_BUNDLES sub machine

```
R1: Execute plantsim
{
  if exec_bundle = plantsim and plantsim_s != done then
    PLANTSIM_BUNDLE();
}

R2: Pass is done
{
  if exec_bundle = plantsim and plantsim_s = done then
    plantsim_s  := not_done;
    exec_bundle := plantsim;
    execution   := done;
}
```

Listing F.9 Rules of the PLANTSIM_BUNDLE sub machine

```
R1: Bundle Active
{
  if plantsim_bundle_status = active then
    EXECUTE_PLANTSIM_SEQUENCES();
}

R2: Bundle Inactive
{
  if plantsim_bundle_status = inactive then
    plantsim_s := done;
}
```

Listing F.10 Rules of the EXECUTE_PLANTSIM_SEQUENCES sub machine

```
R1: Exec Temp Seq
{
  if exec_seq = temp_monitor and temp_seq_s = not_done then
    SEQUENCE_TEMP_MONITOR();
}

R2: Switch Seq
{
  if exec_seq = temp_monitor and temp_seq_s = done then
    exec_seq    := humid_monitor;
    humid_seq_s := not_done;
}

R3: Exec Humid Seq
{
  if exec_seq = humid_monitor and humid_seq_s = not_done then
    SEQUENCE_HUMIDITY_MONITOR();
}

R4: Bundle finished
{
  if exec_seq = humid_monitor and humid_seq_s = done then
    plantsim_s := done;
    exec_seq   := temp_monitor;
    temp_seq_s := not_done;
}
```

Listing F.11 Rules of the SEQUENCE_TEMP_MONITOR sub machine

```
R1: Sequence Active
{
  if temp_seq_status = active then
    SEQUENCE_TEMP_MONITOR_WORK();
}

R2: Sequence Inactive
{
  if temp_seq_status = inactive then
    temp_seq_s := done;
}
```

Listing F.12 Rules of the SEQUENCE_TEMP_MONITOR_WORK sub machine (part 1)

```
R1: b0 -> b1
{
  t := 685;

  if temp_seq_b = b0 then
    temp_seq_b := b1;
}

R2: b1 -> b2
{
  t := 2285;

  if temp_seq_b = b1 and temperature >= 26 then
    temp_seq_b              := b2;
    trying_to_cool_system   := True;
    cooling                 := on;
}

R3: b1 -> b3
{
  t := 1730;

  if temp_seq_b = b1 and temperature < 26 then
    temp_seq_b := b3;
}

R4: b2 -> b2
{
  t := 1625;

  if temp_seq_b = b2 and temperature > 22 then
    temp_seq_b := b2;
    temp_seq_s := done;
}
```

Listing F.13 Rules of the SEQUENCE_TEMP_MONITOR_WORK sub machine (part 2)

```
R5: b2 -> b3
{
  t := 1730;

  if temp_seq_b = b2 and temperature <= 22 then
    temp_seq_b             := b3;
    trying_to_cool_system  := False;
    cooling                := off;
}

R6: b3 -> b0
{
  t := 1950;

  if temp_seq_b = b3 and temperature > 19 then
    temp_seq_b := b0;
    temp_seq_s := done;
}

R7: b3 -> b4
{
  t := 2390;

  if temp_seq_b = b3 and temperature <= 19 then
    temp_seq_b  := b4;
    heating     := on;
}

R8: b4 -> b4
{
  t := 1630;

  if temp_seq_b = b4 and temperature < 22 then
    temp_seq_b := b4;
    temp_seq_s := done;
}

R9: b4 -> b0
{
  t := 3195;

  if temp_seq_b = b4 and temperature >= 22 then
    temp_seq_b  := b0;
    heating     := off;
    temp_seq_s  := done;
}
```

Listing F.14 Rules of the SEQUENCE_HUMIDITY_MONITOR sub machine

```
R1: Sequence Active
{
  if humid_seq_status = active then
    SEQUENCE_HUMIDITY_MONITOR_WORK();
}

R2: Sequence Inactive
{
  if humid_seq_status = inactive then
    humid_seq_s := done;
}
```

Listing F.15 Rules of the SEQUENCE_HUMIDITY_MONITOR_WORK sub machine (part 1)

```
R1: c0 -> c1
{
  t := 685;

  if humid_seq_b = c0 then
    humid_seq_b := c1;
}

R2: c1 -> c2
{
  t := 2395;

  if humid_seq_b = c1 and humidity >= 61 then
    humid_seq_b             := c2;
    cooling                 := on;
}

R3: c1 -> c4
{
  t := 1730;

  if humid_seq_b = c1 and humidity < 61 then
    humid_seq_b := c4;
}

R4: c2 -> c2
{
  t := 1625;

  if humid_seq_b = c2 and humidity > 50 then
    humid_seq_b := c2;
    humid_seq_s := done;
}

R5: c2 -> c3
{
  t := 1625;

  if humid_seq_b = c2 and humidity <= 50 then
    humid_seq_b := c3;
}

R6: c3 -> c4
{
  t := 2160;

  if humid_seq_b = c3 and trying_to_cool_system = False then
    humid_seq_b := c4;
    cooling     := off;
}
```

Listing F.16 Rules of the SEQUENCE_HUMIDITY_MONITOR_WORK sub machine (part 2)

```
R7: c3 -> c4
{
  t := 1265;

  if humid_seq_b = c3 and trying_to_cool_system = True then
    humid_seq_b  := c4;
}

R8: c4 -> c5
{
  t := 2390;

  if humid_seq_b = c4 and humidity <= 39 then
    humidifier := on;
    humid_seq_b := c5;
}

R9: c4 -> c0
{
  t := 1950;

  if humid_seq_b = c4 and humidity > 39 then
    humid_seq_b := c0;
    humid_seq_s := done;
}

R10: c5 -> c5
{
  t := 1630;

  if humid_seq_b = c5 and humidity < 50 then
    humid_seq_b := c5;
    humid_seq_s := done;
}

R11: c5 -> c0
{
  t := 3195;

  if humid_seq_b = c5 and humidity >= 50 then
    humid_seq_b := c0;
    humid_seq_s := done;
    humidifier:= off;
}
```

THIS PAGE INTENTIONALLY LEFT BLANK

Bibliography

[1] Yasmina Abdeddaïm, Abdelkarim Kerbaa, and Oded Maler. Task Graph Scheduling using Timed Automata. In *Proceedings of the 17th International Parallel and Distributed Processing Symposium (IPDPS '03)*, April 2003.

[2] Jean-Raymond Abrial and Louis Mussat. Specification and Design of a Transmission Protocol by Successive Refinements using B. In *Mathematical Methods in Program Development.* Springer-Verlag, 1997.

[3] Rajeev Alur, Thao Dang, Joel Esposito, Rafael Fierro, Yerang Hur, Franjo Ivancic, Vijay Kumar, Insup Lee, Pradyumna Mishra, George Pappas, and Oleg Sokolsky. Hierarchical Hybrid Modeling of Embedded Systems. In *Proceedings of the 1st Workshop on Embedded Software (EMSOFT '01)*, 2001.

[4] Rajeev Alur, Thao Dang, Joel Esposito, Yerang Hur, Franjo Ivancoc, Vijay Kumar, Insup Lee, Pradyumna Mishra, George Pappas, and Oleg Sokolsky. Hierarchical Modeling and Analysis of Embedded Systems. *Proceedings of the IEEE*, 91(1), January 2003.

[5] Rajeev Alur and David L. Dill. A Theory of Timed Automata. *Theoretical Computer Science*, 126(2), 1994.

[6] Rajeev Alur, Radu Grosu, Yerang Hur, Vijay Kumar, and Insup Lee. Modular Specification of Hybrid Systems in CHARON. In *Proceedings of the 3rd International Workshop on Hybrid Systems: Computation and Control*, 2000.

[7] Rajeev Alur and Parthasarathy Madhusudan. Decision Problems for Timed Automata: A Survey. *Formal Methods for the Design of Real-Time Systems*, 3185:1–24, 2004.

[8] Paul E. Ammann, Paul E. Black, and William Majurski. Using model checking to generate tests from specifications. In *Proceedings of the 2nd IEEE International Conference on Formal Engineering Methods (ICFEM '98)*, page 46, 1998.

[9] Tobias Amnell, Elena Fersman, Leonid Mokrushin, Paul Pettersson, , and Wang Yi. TIMES: a Tool for Schedulability Analysis and Code Generation of Real-Time Systems. In *Proceedings of the 1st International Workshop on Formal Modeling and Analysis of Timed Systems (FORMATS '03)*, 2003.

[10] Matthias Anlauff. XASM - An Extensible, Component-Based Abstract State Machines Language. In *Abstract State Machines - ASM 2000, International Workshop on Abstract State Machines.* TIK-Report 87, 2000.

[11] Myla Archer, Constance Heitmeyer, and Steve Sims. TAME: A PVS Interface to Simplify Proofs for Automata Models. In *Proceedings of the International Workshop on User Interfaces for Theorem Provers (UITP '98)*, 1998.

[12] Myla Archer, Ben Di Vito, and Cesar Munoz. Developing User Strategies in PVS: A Tutorial. In *Proceedings of the 1st International Workshop on Design and Application of Strategies/Tactics in Higher Order Logics (STRATA '03)*, 2003.

[13] Mylan Archer. TAME: Using PVS Strategies for Special-Purpose Theorem Proving. *Annals of Mathematics and Artificial Intelligence*, 29:139–181, 2000.

[14] Alessandro Armando, Jacopo Mantovani, and Lorenzo Platania. Bounded Model Checking of Software Using SMT Solvers Instead of SAT Solvers. In *Proceedings of the 13th International SPIN Workshop (SPIN '06)*, volume 3925 of *LNCS*, pages 146–162, April 2006.

[15] Cyrille Artho, Howard Barringer, Allen Goldberg, Klaus Havelund, Sarfraz Khurshid, Mike Lowry, Corian Pasareanu, Grigore Rosu, Koushik Sen, Willem Visser, and Rich Washington. Combining test case generation and runtime verification. In *Theoretical Computer Science*, volume 336. Elsevier B. V., 2005.

[16] Cyrille Artho, Doron Drusinsky, Allen Goldberg, Klaus Havelund, Mike Lowry, Corina Pasareanu, Grigore Roşu, and Willem Visser. Experiments with Test Case Generation and Runtime Analysis. In *Abstract State Machines, Advances in Theory and Practice, 10th International Workshop, ASM 2003*, volume 2589 of *LNCS*. Springer, March 3–7 2003.

[17] R. J. R. Back. On Correct Refinement of Programs. *Journal of Computer and Systems Sciences*, 23(1):49–68, 1979.

[18] John W. Backus and Peter Naur. Revised Report on the Algorithmic Language ALGOL 60. *Communications of the ACM*, 6(1), 1963.

[19] A. Terry Bahill and Bruce Gissing. Re-Evaluating Systems Engineering Concepts Using Systems Thinking. *IEEE Transaction on Systems, Man and Cybernetics, Part C: Applications and Reviews*, 28(4):516–527, 1998.

[20] John Barnes. *High Integrity Software: The SPARK Approach to Safety and Security*. Addison-Wesley, 2003.

[21] Clark Barrett, Leoinardo de Moura, and Aaron Stump. Design and Results of the 1st Satisfiability Modulo Theories Competition (SMT-COMP '05). *Journal of Automated Reasoning*, 35(4):373–390, 2005.

[22] Daniele Beauquier and Anatol Slissenko. On Verification of Refinements of Asynchronous Timed Distributed Algorithms. In *Abstract State Machines: Theory and Applications*, volume 1912 of *LNCS*, pages 34–39. Springer-Verlag, 2000.

[23] Michael Von Der Beek. A Comparison of Statechart Variants. *Formal Techniques in Real-Time and Fault-Tolerant Systems*, 863, 1994.

[24] Gerd Behrmann, Alexandre David, and Kim G. Larsen. A Tutorial on UPPAAL. In *Proceedings of the 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems (SFM-RT '04)*, volume 3185 of *LNCS*. Springer-Verlag, 2004.

[25] Gerd Behrmann, Kim G. Larsen, and Jacob I. Rasmussen. Optimal Scheduling using Priced Timed Automata. *SIGMETRICS Performance Evaluation Review*, 32(4), 2005.

[26] Boris Beizer. *Software Testing Techniques.* International Thomson Computer Press, 2nd edition, 1990.

[27] Axel Belinfante, Lars Frantzen, and Christian Schallhart. 14 Tools for Test Case Generation. In *Model-Based Testing of Reactive Systems*, volume 3472 of *LNCS*. Springer-Verlag, 2005.

[28] Mordechai Ben-Ari. *Mathematical Logic for Computer Science.* Springer, 2nd edition, 2001.

[29] Johan Bengtsson and Wang Yi. Timed Automata: Semantics, Algorithms and Tools. *Lecture Notes on Concurrency and Petri Nets*, 3098, 2004.

[30] Béatrice Berard, Michel Bidoit, Alain Finkel, François Laroussinie, Antoine Petit, Laure Petruccci, Philippe Schnoebelen, and Pierre McKenzie. *Systems and Software Verification.* Springer-Verlag, 2001.

[31] Jan A. Bergstra, Alban Ponse, and Scott A. Smolka. *A Handbook of Process Algebra.* North-Holland, 2001.

[32] Kirsten Berkenkötter and Raimund Kirner. Real-Time and Hybrid Systems Testing. In *Model-Based Testing of Reactive Systems*, pages 355–387, 2004.

[33] Marco Bernardo and Flavio Corradini. *Formal Methods for the Design of Real-Time Systems.* Springer-Verlag, 2004.

[34] Gerald Berry. The Essence of ESTEREL. In *Proof, Language and Interaction: Essays in Honour of Robin Milner.* MIT Press, 2000.

[35] Gérard Berry, Michael Kishinevsky, and Satnam Singh. System Level Design and Verification Using a Synchronous Language. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD '03)*, 2003.

[36] Johan Blom, Anders Hessel, Bengt Jonsson, and Paul Pettersson. Specifying and Generating Test Cases Using Observer Automata. In *Formal Approaches to Software Testing*, volume 3395 of *LNCS.* Springer Berlin / Heidelberg, 2005.

[37] Barry W. Boehm. *Software Engineering Economics.* Prentice-Hall, 1981.

[38] Barry W. Boehm. Software Engineering Economics. *IEEE Transactions on Software Engineering*, 10(1):4–21, 1984.

[39] Barry W. Boehm. Understanding and Controlling Software Costs. *IEEE Transactions on Software Engineering*, 14(10):1462–1477, 1988.

[40] Barry W. Boehm. Unifying Software Engineering and Systems Engineering. *IEEE Computers*, 33(3):114–116, 2000.

[41] Egon Börger. Why Use Evolving Algebras for Hardware and Software Engineering? In *Proceedings of the 22nd Seminar on Current Trends in Theory and Practice of Informatics (SOFSEM '95)*, volume 1012 of *LNCS.* Springer-Verlag, 1995.

[42] Egon Börger. The Origins and the Development of the ASM Method for High Level System Design and Analysis. *Journal of Computer Science*, 8(5), 2001.

[43] Egon Börger. The ASM Refinement Method. *Formal Aspects of Computing*, 15(2–3):237–257, 2003.

[44] Egon Börger, Yuri Gurevich, and Dean Rosenzweig. The Bakery Algorithm: Yet Another Specification and Verification. In *Specification and Validation Methods*, pages 231–243. Oxford University Press, 1995.

[45] Egon Börger and Luca Mearelli. Integrating ASMs into the Software Development Life Cycle. *Journal of Universal Computer Science*, 3(5), 1997.

[46] Egon Börger, Peter Päppinghaus, and Joachim Schmid. Report on a practical application of ASMs in software design. In *Abstract State Machines: Theory and Applications*, volume 1912. Springer-Verlag, 2000.

[47] Egon Börger and Robert Stärk. *Abstract State Machines.* Springer-Verlag, 2003.

[48] Yves Boussemart, Sébastien Gorelov, Martin Ouimet, and Kristina Lundqvist. Non-Intrusive System-Level Fault Tolerance for an Electronic Throttle Controller. In *Proceedings of the Fifth International Conference on Networking (ICN '06) and International Conference on Systems (ICONS '06) and International Conference on Mobile Communications and Learning.* IEEE Computer Society Press, April 23–29 2006.

[49] Bruno Bouyssounouse and Joseph Sifakis. *Embedded Systems Design: The ARTIST Roadmap for Research and Development.* Springer, 2005.

[50] Jonathan P. Bowen and Michael G. Hinchey. Seven More Myths of Formal Methods. *IEEE Software*, 12(4), July 1994.

[51] Jonathan P. Bowen and Michael G. Hinchey. Ten Commandments of Formal Methods. *IEEE Computers*, 28(4), April 1994.

[52] Marius Bozga, Susanne Graf, Ileana Ober, Iulian Ober, and Joseph Sifakis. The IF Toolset. In *Formal Methods for the Design of Real-Time Systems*, volume 3185, pages 237–267. LNCS, 2004.

[53] Marco Bozzano, Roberto Bruttomesso, Alessandro Cimatti, Tommi Junttila, Peter Rossum, Stephan Schulz, and Roberto Sebastiani. MathSAT: Tight Integration of SAT and Mathematical Decision Procedures. *Journal of Automated Reasoning*, 35(1–3):265–293, 2005.

[54] Barrett R. Bryant. Object-Oriented Natural Language Requirements Specification. In *Proceedings of the 23rd Australasian Computer Science Conference (ASCS '00)*, pages 24–30, 2000.

[55] Joseph Buck, Soonhoi Ha, Edward A. Lee, and David G. Messerschitt. Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems. *International Journal of Computer Simulation*, 4, April 1994.

[56] David Budgen. *Software Design*. Addison Wesley, 2nd edition, 2003.

[57] Alan Burns. The Ravenscar Profile. White Paper available from http://polaris.dit.upm.es/ ork/documents.

[58] Alan Burns and Andy Wellings. *Real-Time Systems and Programming Languages*. Addison-Wesley, 2001.

[59] Giuseppe Del Castillo. Towards Comprehensive Tool Support for Abstract State Machines: The ASM Workbench Tool Environment and Architecture. In *Applied Formal Methods – FM-Trends 98*, volume 1641 of *LNCS*, pages 311–325. Springer, 1999.

[60] Antonio Cerone and Andrea Maggiolo-Schettini. Time-based Expressivity of Time Petri Nets for System Specification. In *Theoretical Computer Science*, volume 216. Springer-Verlag, 1999.

[61] Charles Stark Draper Laboratory. The Timeliner User Interface Language (UIL) System for the International Space Station. Technical Report available from http://timeliner.draper.com.

[62] Charles Stark Draper Laboratory. User Interface Language CPU Performance Report for the ISS Timeliner. Technical Report CSDL 306647, September 1997. Available from http://timeliner.draper.com.

[63] Alan K. Cheng. *Real-Time Systems: Schedulability, Analysis, and Verification.* John Wiley and Sons, 2003.

[64] John Joseph Chilenski and Steven P. Miller. Applicability of Modified Condition/Decision Coverage Software Testing. *Software Engineering Journal*, September 1994.

[65] Alessandro Cimatti, Edmund M. Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. NuSMV 2: An OpenSource Tool for Symbolic Model Checking. In *Proceeding of the 14th International Conference on Computer-Aided Verification (CAV '02)*, volume 2404 of *LNCS*. Springer-Verlag, 2002.

[66] Marcus Ciolkowski, Oliver Laitenberger, Dieter Rombach, Forrest Shull, and Dewayne Perry. Software Inspections, Reviews, and Walkthroughs. In *Proceedings of the 15th Internation Conference on Software Engineering (ICSE '02)*, 2002.

[67] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic Verification of Finite State Concurrent System Using Temporal Logic Specifications: a Practical Approach. In *Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of programming languages (POPL '83)*, pages 117–126. ACM Press, 1983.

[68] Edmund Clarke, Daniel Kroening, Joel Ouaknine, and Ofer Strichman. Computational Challenges in Bounded Model Checking. *Software Tools for Technology Transfer*, 7(2):174–183, April 2005.

[69] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Progress on the State Explosion Problem in Model Checking. In *Informatics*, volume 2000 of *LNCS*, pages 176–194. Springer-Verlag, 2001.

[70] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, 1999.

[71] Edmund M. Clarke and Jeannette Wing. Formal Methods: State of the Art and Future Directions. *ACM Computing Surveys*, 28(3), 1996.

[72] Joëlle Cohen and Anatol Slissenko. On Verification of Refinements of Asynchronous Timed Distributed Algorithms. In *International Workshop on Abstract State Machines (ASM '00)*, pages 100–114. Springer-Verlag, 2000.

[73] Kendra Cooper and Mabo Ito. Formalizing a Structured Natural Language Requirements Specification Notation. In *Proceedings of the INCOSE 2002 Conference, Engineering 21st Century Systems: Problem Solving Through Structured Thinking*, 2002.

[74] Patrick Cousot and Radhia Cousot. Abstract Interpretation Based Program Testing. In *Proceedings of the SSGRR 2000 Computer & eBusiness International Conference*, July 31 – August 6 2000.

[75] Steven J. Cunning and Jerzy W. Rozenblit. Test Scenario Generation from a Structured Requirements Specification. In *Proceedings of the Sixth IEEE Conference and Workshop on Engineering of Computer-Based Systems (ECBS '99)*, March 7–12 1999.

[76] Bharat P. Dave, Ganesh Lakshminarayana, and Niraj K. Jha. COSYN: Hardware-Software Co-Synthesis of Embedded Systems. In *Design Automation Conference*, pages 703–708, 1997.

[77] Weillem P. de Roever and Kai Engelhardt. *Data Refinement: Model-Oriented Proof Methods and their Comparison.* Cambridge University Press, 1998.

[78] Department of Defense. Military Handbook: Configuration Management Guidance, revision A. DoD Report MIL-HDBK-61A, 2001.

[79] John Derrick and Eerke Boiten. Refinement in Z and Object-Z. In *Formal Approaches to Computing and Information Technology.* Springer-Verlag, 2001.

[80] Edsger W. Dijkstra. Notes on Structured Programming. In *Structured Programming*, pages 1–82. Academic Press, 1972.

[81] Edsger W. Dijkstra. *A Discipline of Programming.* Prentice-Hall Series in Automatic Computation, 1976.

[82] Antoni Diller. *Z: An Introduction to Formal Methods.* John Wiley and Sons, Inc., 1990.

[83] Gilles Dowek, Amy Felty, Gérard Huet, Hugo Herbelin, Chet Murthy, Catherine Parent, Christine Paulin-Mohring, and Benjamin Werner. The Coq Proof Assistant User's Guide. Technical Report, INRIA, 1993. Available from http://coq.inria.fr.

[84] David Duffy. *Principles of Automated Theorem Proving.* John Wiley and Sons, 1991.

[85] Douglas D. Dunlop and Victor R. Basili. A Comparative Analysis of Functional Correctness. *ACM Computing Surveys (CSUR)*, 14:229–244, 1982.

[86] Jon Edvardson. A survey on automatic test data generation. In *Proceedings of the 2nd Conference on Computer Science and Engineering in Linköping*, 1999.

[87] Niklas Eén and Niklas Sörensson. An Extensible SAT Solver. In *Proceedings of the 6th International Conference on Theory and Applications of Satisfiability Testing*, volume 2919 of *LNCS*, pages 502–518, 2003.

[88] Embedded Systems Laboratory. The Timed Abstract State Machine Language and Toolset. http://esl.mit.edu/tasm.

[89] Jakob Engblom, Andreas Ermedahl, Mikael Nolin, Jan Gustafsson, and Hans Hansson. Worst-Case Execution-Time Analysis for Embedded Real-Time Systems. *International Journal on Software Tools for Technology Transfer*, 4:437–455, October 2003.

[90] Jakob Engblom, Andreas Ermedahl, and Friedhelm Stapper. A Worst-Case Execution-Time Analysis Tool Prototype for Embedded Real-Time Systems.

In *Proceedings of the 1st Workshop on Real-Time Tools (RT-TOOLS '01), held in conjunction with CONCUR '01*, August 2001.

[91] Edwin Erpenbach, Friedhelm Stappert, and Joachim Stroop. Compilation and Timing Analysis of Statecharts Models for Embedded Systems. In *The 2nd International Workshop on Compiler and Architecture Support for Embedded Systems (CASES'99)*, 1999.

[92] Robert Eschbach, Uwe Glässer, Reinhard Gotzhein, and Andreas Prinz. On the Formal Semantics of Design Languages: A Compilation Approach using Abstract State Machines. In *Proceedings of the International Workshop on Abstract State Machines - ASM 2000*, volume 1912 of *LNCS*. Springer-Verlag, 2000.

[93] Huáscar Espinoza, Hubert Dubois, Julio Medina, and Sébastien Gérard. A General Structure for the Analysis Framework of the UML MARTE Profile. In *Proceedings of the International Workshop on Modeling and Analysis of Real-Time and Embedded Systems (MARTES '05)*, 2005.

[94] Yaodong Feng, Gang Huang, Jie Yang, and Hong Mei Mei. Traceability between Software Architecture Models. In *Proceedings of the 30th Annual International Computer Software and Applications Conference (COMPSAC '06)*, pages 41–44. IEEE Computer Society, 2006.

[95] Wan Fokkink. *Introduction to Process Algebra*. Springer-Verlag, 2000.

[96] Free Software Foundation. GNU Linear Programming Kit (GLPK). Available from http://www.gnu.org/software/glpk/.

[97] Martin Fränzle. Verification of Hybrid Systems. In *Proceedings of the 19th International Conference on Computer Aided Verification (CAV '07)*, volume 4590 of *LNCS*, page 38, 2007.

[98] Norbert E. Fuchs. Specifications are (Preferably) Executable. *IEE/BCS Software Engineering Journal*, 7(5), 1992.

[99] Roger Fujii and Dolores R. Wallace. Software Verification and Validation: An Overview. *IEEE Software*, 6(3):220–234, 1989.

[100] Angelo Furfaro and Libero Nigro. Model Checking Time Petri Nets: A Translation Approach based on UPPAAL and a Case Study. In *Proceedings of the IASTED International Conference on Software Engineering (SE '05)*, pages 388–393, 2005.

[101] Peter Galison. *Einstein's Clocks, Poincare's Maps: Empires of Time.* W. W. Norton and Company, 2004.

[102] Angelo Gargantini and Constance Heitmeyer. Using Model Checking to Generate Tests from Requirements Specifications. In *Proceedings of the 7th European software engineering conference held jointly with the 7th ACM SIGSOFT international symposium on Foundations of software engineering (ESEC/FSE-7)*, pages 146–162, 1999.

[103] Angelo Gargantini and Elvinia Riccobene. Encoding Abstract State Machines in PVS. In *Proceedings of the International Workshop on Abstract State Machines – ASM 2000*, volume 1912 of *LNCS*. Springer-Verlag, 2000.

[104] Angelo Gargantini and Elvinia Riccobene. ASM-based Testing: Coverage Criteria and Automatic Test Sequence Generation. *Journal of Universal Computer Science*, 7(11), 2001.

[105] Abdelouahed Gherbi and Ferhat Khendek. UML Profiles for Real-Time Systems and their Applications. *Journal of Object Technology*, 5(4), May-June 2006.

[106] M. J. C. Gordon and T. F. Melham. *HOL: A Theorem Proving Environment for Higher Order Logic.* Cambridge University Press, 1993.

[107] Susanne Graf and Ileana Ober. How useful is the UML profile SPT without Semantics? In *Proceedings of the International Workshop on Specification, Implementation and Validation of Object-Oriented Embedded Systems (SIVOES 2004)*, 2004.

[108] Susanne Graf, Ileana Ober, and Iulian Ober. A Real-Time Profile for UML. *International Journal on Software Tools for Technology Transfer*, 8(2):113–127, 2006.

[109] W. Grieskamp, L. Nachmanson, N. Tillmann, and M. Veanes. Test Case Generation from AsmL Specifications - Tool Overview. In *Abstract State Machines – ASM 2003*. Springer-Verlag, 2003.

[110] Wolfgang Grieskamp, Yuri Gurevich, Wolfram Schulte, and Margus Veanes. Generating Finite State Machines From Abstract State Machines. In *Proceedings of the 2002 ACM SIGSOFT international symposium on Software Testing and Analysis*, pages 112–122, 2002.

[111] Paul G. Griffiths. Embedded Software Control Design for an Electronic Throttle Body. Master's thesis, University of California, Berkeley, 2002.

[112] Pallav Gupta, Steven J. Cunning, and Jerzy W. Rozenblit. Synthesis of High-Level Requirements Models for Automated Test Generation. In *Proceedings of the Eighth Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS '01)*, April 17–20 2001.

[113] Yuri Gurevich. Evolving Algebras 1993: Lipari Guide. *Specification and Validation Methods*, pages 9–36, 1995.

[114] Yuri Gurevich and Jim Huggins. The Railroad Crossing Problem: an Experiment with Instantaneous Actions and Immediate Reactions. In H. K. Buening, editor, *Lecture Notes in Computer Science, Computer Science Logics, Selected Papers from CSL '95*, volume 1092 of *LNCS*, pages 266–29. Springer-Verlag, 1996.

[115] Yuri Gurevich and Dean Rosenzweig. Partially Ordered Runs: A Case Study. In *Abstract State Machines: Theory and Applications*, volume 1912 of *LNCS*, pages 131–150. Springer-Verlag, 2000.

[116] Yuri Gurevich, Bejamin Rossman, and Wolfram Schulte. Semantic Essence of AsmL. *Theoretical Computer Science*, 343(3):370–412, 2005.

[117] Anthony Hall. Seven myths of formal methods. *IEEE Software*, 07(5):11–19, 1990.

[118] Anthony Hall and Roderick Chapman. Correctness by Construction: Developing a Commercial Secure System. *IEEE Software*, 19(1), 2002.

[119] Kaj Hänninen, Jukka Mäki-Turja, Markus Bohlin, Jan Carlson, and Mikael Nolin. Determining Maximum Stack Usage in Preemptive Shared Stack Systems. In *Proceedings of the 27th Real-Time System Symposium (RTSS '06)*, pages 445–453, 2006.

[120] David Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 1987.

[121] David Harel, Hagi Lachover, Amnon Naamad, Amir Pnueli, Michal Politi, Rivi Sherman, and Aharon Shtul-Trauring. STATEMATE: a Working Environment for the Development of Complex Reactive Systems. In *Proceedings of the 10th International Conference on Software Engineering (ICSE '88)*, pages 396–406. IEEE Computer Society Press, 1988.

[122] David Harel and Amnon Naamad. The STATEMATE Semantics of Statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4), 1996.

[123] Mats Per Erik Heimdahl and Nancy G. Leveson. Completeness and Consistency in Hierarchical State-Based Requirements. *Software Engineering*, 22(6):363–377, 1996.

[124] Constance L. Heitmeyer. On the Need for Practical Formal Methods. In *Proceedings of the 5th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT '98)*, pages 18–26. Springer-Verlag, 1998.

[125] Constance L. Heitmeyer, Raplh D. Jeffords, and Bruce G. Labaw. Automated Consistency Checking of Requirements Specifications. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 5(3):231–261, July 1996.

[126] Keijo Heljanko, Tommi Junttila, and Timo Latvala. Incremental and Complete Bounded Model Checking for Full PLTL. In *Proceedings of the 17th International Conference on Computer Aided Verification (CAV '05)*, volume 3576 of *LNCS*, pages 98–111. Springer-Verlag, July 2005.

[127] Thomas A. Henzinger, Pei-Hsin Ho, and Howard Wong-Toi. HyTech: The Next Generation. In *Proceedings of the IEEE Real-Time Systems Symposium (RTSS '95)*, pages 56–65, 1995.

[128] Anders Hessel, Kim G. Larsen, Brian Nielsen, Paul Pettersson, and Arne Skou. Time-Optimal Real-Time Test Case Generation Using UPPAAL . *Formal Approaches to Software Testing*, 2931:114–130, 2004.

[129] Anders Hessel and Paul Pettersson. A Test Case Generation Algorithm for Real-Time Systems. In *Proceedings of the Fourth International Conference on Quality Software (QSIC '04)*, 2004.

[130] Robert Hierons and John Derrick. Special Issue on Specification-Based Testing. *Software Testing, Verification, and Reliability*, 10:201–262, 2000.

[131] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice Hall, 1985.

[132] C.A.R. Hoare. Assertions: A Personal Perspective. *IEEE Annals of the History of Computing*, 25(2):14–25, 2003.

[133] C.A.R. Hoare. Process algebra: A unifying approach. In *25 years of Communicating Sequential Processes.* Springer-Verlag, 2005.

[134] John Holt. *UML for Systems Engineering - Watching the Wheels.* IEE Professional Applications of Computing Series Volume 4, 2nd edition, 2004.

[135] Gerard J. Holzmann and Rajeev Joshi. Model-Driven Software Verification. In *Proceedings of the 11th Spin Workshop*, volume 2989 of *LNCS*, pages 77–92. Springer-Verlag, 2004.

[136] Jim Huggins. Kermit: Specification and Verification. In *Specification and Validation Methods*, pages 247–293. Oxford University Press, 1995.

[137] Michael A. Huth and Mark D. Ryan. *Logic in Computer Science: Modelling and reasoning about systems.* Cambridge University Press, 2000.

[138] Zbigniew Huzar, Ludwik Kuzniarz, Gianna Reggio, and Jean Louis Sourrouille. Consistency Problems in UML-Based Software Development. In *UML Modeling Languages and Applications*, volume 3297 of *LNCS*, pages 1–12, 2005.

[139] International Standars Organization: TC 176, SC 2. ISO 10007: Quality Management Systems Guidelines for Configuration Management. American National Standards Institute (ANSI), 2007.

[140] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis* . MIT Press, 2006.

[141] Michael Jackson. *Software Requirements & Specifications.* Addison-Wesley, 1995.

[142] Paul B. Jackson, Bill J. Ellis, and Kathleen Sharp. Using SMT Solvers to Verify High-Integrity Programs. In *Proceedings of the 2nd Workshop on Automated Formal Methods (AFM '07)*, volume 3576, pages 60–68, 2007.

[143] Lalita Jategaonkar Jagadeesan, Lawrence G. Votta, Adam Porter, Carlos Puchol, and J. Christopher Ramming. Specification-Based Testing of Reactive Software: a Case Study in Technology Transfer. *Journal of Systems and Software*, 40(3):249–262, 1998.

[144] Axel Jantsch and Ingo Sander. Models of Computation and Languages for Embedded System Design. *IEE Proceedings - Computers and Digital Techniques*, 152(2), March 2005.

[145] Luc Jaulin, Michel Kieffer, Olivier Didrit, and Eric Walter. *Applied Interval Analysis.* Springer, 2001.

[146] Chris W. Johnson. Literate Specifications. *Software Engineering Journal*, 11(4), July 1996.

[147] Dilsun K. Kaynar, Nancy Lynch, Roberto Segala, and Frits Vaandrager. *The Theory of Timed I/O Automata.* Synthesis Lectures on Computer Science. Morgan & Claypool Publishers, 2006.

[148] R. Baker Kearfott and Vladik Kreinovich (Eds.). *Applications of Interval Computations.* Kulwer Academic Press, 1996.

[149] Sarfraz Khurshid and Darko Marinov. TestEra: A Novel Framework for Automated Testing of Java Programs. In *Proceedings of the 16th International Conference on Automated Software Engineering (ASE '01)*, November 2001.

[150] Raimund Kirner and Peter Puschner. Classification of WCET Analysis Techniques. In *Proceedings of the 8th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC '05).* IEEE Computer Society, 2005.

[151] Donald E. Knuth. Literate Programming (1984). *Literate Programming*, 1992.

[152] Alexander Kossiakoff and William N. Sweet. *Systems Engineering Principles and Practice.* Wiley-Interscience, 2002.

[153] Richard Kuhn. Fault Classes and Error Detection Capability of Specification-Based Testing. *ACM Transactions on Software Engineering and Methodology*, 8:411–424, 1999.

[154] Leslie Lamport and Fred B. Schneider. Formal Foundation for Specification and Verification. *Distributed Systems: methods and tools for specification: an advanced course*, 190:203–285, 1985.

[155] Phillip A. Laplante. *Real-Time Systems Design and Analysis.* John Wiley and Sons, Inc., 3rd edition, 2004.

[156] Kim G. Larsen, Paul Pettersson, and Wang Yi. UPPAAL : Status and Developments. In *Computer Aided Verification (CAV '97)*, volume 1254 of *LNCS*, pages 456–459, 1997.

[157] Kim Guldstrand Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a Nutshell. *International Journal on Software Tools for Technology Transfer*, 1:134–152, 1997.

[158] Daniel Leberre. SAT4J : A Satisfiability Library for Java. presentation available from http://sat4j.objectweb.com.

[159] Luc Leonard and Guy Leduc. A Formal Definition of Time in LOTOS. *Formal Aspects of Computing*, 10, 1998.

[160] Nancy Leveson, Jon D. Reese, and Mats P. Heimdahl. SpecTRM: A CAD System for Digital Automation. In *Proceedings of the Digital Avionics System Conference (DASC '98)*, 1998.

[161] Nancy G. Leveson. Intent Specifications: An Approach to Building Human-Centered Specifications. *Software Engineering*, 26(1):15–35, 2000.

[162] Nancy G. Leveson, Mats P. Heimdahl, and Jon D. Reese. Designing Specification Languages for Process Control Systems: Lessons Learned and Steps to the Future. In *Proceedings of the 7th European Software Engineering Conference (ESEC '99)*, pages 127–145, 1999.

[163] Claus Lewerentz and Thomas Lindner. Production Cell: A Comparative Study in Formal Specification and Verification. In *KORSO - Methods, Languages, and Tools for the Construction of Correct Software*, 1995.

[164] Jane Liu. *Real-Time Systems.* John Wiley and Sons, 2001.

[165] Nancy Lynch, Laurent Michel, and Alexander Shvartsman. Tempo: A Toolkit for The Timed Input/Output Automata Formalism. In *SimulationWorks*, 2008.

[166] Mark W. Maier and Eberhardt Rechtin. *The Art of Systems Architecting.* CRC, 2nd edition, 2000.

[167] Mathworks. Simulink and Stateflow Product Web Page. http://www.mathworks.com/products/simulink.

[168] Nenad Medvidovic, Paul Grünbacher, Alexander Egyed, and Barry W. Boehm. Bridging Models across the Software Lifecycle. *Journal of Systems and Software*, 68(3):199–215, 2003.

[169] Bertrand Meyer. *Object-oriented software construction.* Prentice Hall, second edition edition, 1997.

[170] Robin Milner. *Communication and Concurrency.* Prentice Hall, 1980.

[171] Sayan Mitra, Myla Archer, HongPing Lim, Nancy Lynch, and Shinya Umeno. Specifying and Proving Properties of Timed I/O Automata in the TIOA Toolkit. In *Proceedings of the 4th ACM-IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE '06)*, 2006.

[172] A. Mitschele-Thiel. *Systems Engineering with SDL.* John Wiley and Sons, Inc., 2001.

[173] Jean-Francois Monin. *Understanding Formal Methods.* Springer-Verlag London Limited, 2003.

[174] J. M. Morris. A Theoretical Basis for Stepwise Refinement. *Science of Computer Programming*, 9(3), 1987.

[175] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of the 38th Design Automation Conference (DAC '01)*, 2001.

[176] Glenford J. Myers. *The Art of Software Testing.* John Wiley and Sons, 2nd edition, 2004.

[177] National Aeronautics and Space Administration (NASA). User Interface Language Specification. NASA Document SSP 30529.

[178] Julian Ober, Susanne Graf, and Ileana Ober. Validating Timed UML Models by Simulation and Verification. *International Journal on Software Tools for Technology Transfer*, 8(2):128–145, 2006.

[179] Object Management Group, Inc. MDA Guide Version 1.0.1. OMG Specification, June 2003.

[180] Object Management Group, Inc. UML Profile for Schedulability, Performance, and Time Specification. OMG Specification, January 2005.

[181] Object Management Group, Inc. Unified Modeling Language: Superstrucure. Version 2.0. OMG Specification, August 2005.

[182] Object Management Group, Inc. UML Profile for Modeling and Analysis of Real-Time and Embedded Systems. OMG Specification, May 2006.

[183] Object Technology International, Inc. Eclipse Platform Overview. Available from http://www.eclipse.org/whitepapers, 2003.

[184] A. Jefferson Offutt, Yiwei Xiong, and Shaoying Liu. Criteria for Generating Specification-based Tests. In *Proceedings of the 5th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS '99)*, pages 119–131, 1999.

[185] Martin Ouimet. The TASM Language Reference Manual, Version 1.3. Available from http://esl.mit.edu/tasm, November 2007.

[186] Martin Ouimet. The TASM Toolset User Manual, Version 1.1. Available from http://esl.mit.edu/tasm, October 2007.

[187] Martin Ouimet, Guillaume Berteau, and Kristina Lundqvist. Modeling an Electronic Throttle Controller using the Timed Abstract State Machine Language and Toolset. In *Proceedings of the 5th International Workshop on Critical Systems Development Using Modeling Languages (CSDUML '06)*, 2006.

[188] Martin Ouimet, Guillaume Berteau, and Kristina Lundqvist. Modeling an Electronic Throttle Controller using the Timed Abstract State Machine Language and Toolset. In *Modeling in Software Engineering*, volume 4364 of *LNCS*, 2007.

[189] Martin Ouimet and Kristina Lundqvist. The Hi-Five Framework and the Timed Abstract State Machine Language. In *Proceedings of the 27th IEEE Real-Time Systems Symposium - Work in Progress Session*, December 2006.

[190] Martin Ouimet and Kristina Lundqvist. A mapping between the Timed Abstract State Machine Language and UPPAAL 's Timed Automata, January 2007. Technical Report ESL-TIK-000211, Embedded Systems Laboratory, Massachusetts Institute of Technology.

[191] Martin Ouimet and Kristina Lundqvist. Automated Theorem Proving: Underlying Theory, Overarching Concepts, and Human Factor Issues, May 2007. Technical Report ESL-TIK-000213, Embedded Systems Laboratory, Massachusetts Institute of Technology.

[192] Martin Ouimet and Kristina Lundqvist. Automated Verification of Completeness and Consistency of Abstract State Machine Specifications using a SAT Solver. In *Proceedings of the 3rd International Workshop on Model-Based Testing (MBT '07), Satellite Workshop of ETAPS '07*, April 2007.

[193] Martin Ouimet and Kristina Lundqvist. Automated Verification of Completeness and Consistency of Abstract State Machine Specifications using a SAT Solver. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 190(2):85–97, 2007.

[194] Martin Ouimet and Kristina Lundqvist. Incorporating Time in the Modeling of Hardware and Software Systems: Concepts, Paradigms, and Paradoxes. In *Proceedings of the International Workshop on Modeling in Software Engineering (MiSE '07), Satellite Workshop of ICSE '07*, May 2007.

[195] Martin Ouimet and Kristina Lundqvist. Modeling the Production Cell System in the TASM Language, January 2007. Technical Report ESL-TIK-000209, Embedded Systems Laboratory, Massachusetts Institute of Technology.

[196] Martin Ouimet and Kristina Lundqvist. The Hi-Five Framework Approach to Reliable Validation of Early System Designs: Bi-Directional Traceability. In *Proceedings of the ARTIST Workshop on Tool Platforms for Embedded System Modeling, Analysis and Validation, Satellite Event of the 19th International Conference on Computer-Aided Verification (CAV '07)*, July 2007.

[197] Martin Ouimet and Kristina Lundqvist. The TASM Language and Toolset: Specification, Validation, and Verification of Embedded Real-Time Systems. In *Presentation Session of the Real-Time in Sweden (RTiS) Conference*, August 2007.

[198] Martin Ouimet and Kristina Lundqvist. The Timed Abstract State Machine Language: Abstract State Machines for Real-Time System Engineering. In *Proceedings of the 14th International Workshop on Abstract State Machines (ASM '07)*, June 2007.

[199] Martin Ouimet and Kristina Lundqvist. The Timed Abstract State Machine Language: An Executable Specification Language for Reactive Real-Time Systems. In *Proceedings of the 15th International Conference on Real-Time and Network Systems (RTNS '07)*, March 2007.

[200] Martin Ouimet and Kristina Lundqvist. The Timed Abstract State Machine Toolset: Specification, Simulation, and Verification of Real-Time Systems. In *Proceedings of the 19th International Conference on Computer-Aided Verification (CAV '07)*, July 2007.

[201] Martin Ouimet and Kristina Lundqvist. Verifying Execution Time using the TASM Toolset and UPPAAL , January 2007. Technical Report ESL-TIK-000212, Embedded Systems Laboratory, Massachusetts Institute of Technology.

[202] Martin Ouimet and Kristina Lundqvist. Bi-Directional Traceability of Software Models: Combining Semantic Refinement and Syntactic Change, January 2008. Technical Report ESL-TIK-000216, Embedded Systems Laboratory, Massachusetts Institute of Technology.

[203] Martin Ouimet and Kristina Lundqvist. The Hi-Five Framework: A Formal Framework for Specification-Based Embedded Real-Time System Engineering. In *Proceedings of the International Symposium on Quality Engineering for Embedded Systems (QEES '08)*, June 2008.

[204] Martin Ouimet and Kristina Lundqvist. The TASM Language and the Hi-Five Framework: Specification, Validation, and Verification of Embedded Real-Time Systems. In *Poster Session of the 14th International Asia-Pacific Software Engineering Conference (APSEC '08)*, December 2008.

[205] Martin Ouimet and Kristina Lundqvist. The Timed Abstract State Machine Language: Abstract State Machines for Real-Time System Engineering. *Journal of Universal Computer Science*, 14(6), July 2008.

[206] Martin Ouimet, Mathieu Quenot, and Kristina Lundqvist. Mixed Integer Programming for Automated Testing and Automated Verification of System Specifications, July 2007. Technical Report ESL-TIK-000215, Embedded Systems Laboratory, Massachusetts Institute of Technology.

[207] Sam Owre, John Rushby, Natarajan Shankar, and David Stringer-Calvert. PVS: An Experience Report. In *Proceedings of Applied Formal Methods–FM Trends 98: The International Workshop on Current Trends in Applied Formal Methods*, volume 1641 of *LNCS*. Springer-Verlag, 1998.

[208] David L. Parnas and Jan Madey. Functional Documents for Computer Systems. In *Science Of Programming*. Elsevier, 1995.

[209] Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *LNCS*. Springer-Verlag, 1991.

[210] Lawrence C. Paulson. The Foundation of a Generic Theorem Prover. *Journal of Automated Reasoning*, 5(3), 1998.

[211] Paul Pettersson and Kim G. Larsen. Uppaal2k. *Bulletin of the European Association for Theoretical Computer Science*, 70:40–44, 2000.

[212] Amir Pnueli. The temporal semantics of concurrent programs. In *Proceedings of the International Symposium on Semantics of Concurrent Computation*, pages 1–20, 1979.

[213] Alexander Pretschner and Heiko Lötzbeyer. Model Based Testing with Constraint Logic Programming: First Results and Challenges. In *Proceedings of the 2nd ICSE International Workshop on Automated Program Analysis, Testing and Verification (WAPATV '01)*, May 2001.

[214] Alexander Pretschner and Jan Philipps. Methodological Issues in Model-Based Testing. In *Model-Based Testing of Reactive Systems*, pages 281–291, 2004.

[215] Jean-François Puget and Irvin Lustig. Constraint Programming and Maths Programming. *The Knowledge Engineering Review*, 16:5–23, 2001.

[216] Radio Technical Commission for Aeronautics (RTCA). Software Considerations in Airborne Systems and Equipment Certification. Document DO-178B.

[217] Balasubramaniam Ramesh and Matthias Jarke. Toward Reference Models for Requirements Traceability. *IEEE Transactions on Software Engineering*, 27(1):58–93, 2001.

[218] Arthur Richards and Jonathan How. Mixed-integer Programming for Control. In *Proceedings of the 2005 American Control Conference (ACC '05)*, June 2005.

[219] Debra J. Richardson, Stephanie Leif Aha, and T. Owen O'Malley. Specification-Based Test Oracles for Reactive Systems. In *Proceedings of the 14th International Conference on Software Engineering (ICSE '92)*, pages 105–118, 1992.

[220] David S. Rosenblum. Formal Methods and Testing: Why the State-of-the Art in Not the State-of-the Practice. *ACM SIGSOFT Software Engineering Notes*, 21(4), July 1996.

[221] Heinrich Rust. Using Abstract State Machines: Using the Hypperreals for Describing Continuous Changes in a Discrete Notation. In *Abstract State Machines - ASM 2000.* Springer-Verlag, March 2000.

[222] Matthew J. Rutherford and Alexander L. Wolf. A Case for Test-Code Generation in Model-Driven Systems. In *Proceedings of the 2nd International Conference on Generative Programming and Component Engineering*, March 7–12 2003.

[223] SAE Aerospace. *Architecture Analysis & Design Language Standard.* SAE Publication AS506, 2004.

[224] Motoshi Saeki, Hisayuki Horai, and Hajime Enomoto. Software Development from Natural Language Specification. In *Proceedings of the 11th International Conference on Software Engineering (ICSE '89)*, pages 64–73, 1989.

[225] Hossein Saiedian. An invitation to formal methods. *Computer*, 29(4):16–30, 1996.

[226] Ahmed M. Salem. Improving Software Quality through Requirements Traceability Models. In *Proceedings of the IEEE International Conference on Computer Systems and Applications*, pages 1159–1162, 2006.

[227] Davide Sangiorgi. On the Bisimulation Proof Method. *Mathematical Structures in Computer Science*, 8(5):447–479, 1998.

[228] Douglas C. Schmidt. Model Driven Development for Distributed Real-Time and Embedded System. In *Proceedings of the 8th Intenational Conference on Model Driven Engineering Languages and Systems (MoDELS '05)*, volume 3713 of *LNCS*. Springer-Verlag, 2005.

[229] Hossein M. Sheini and Karem A. Sakallah. From Propositional Satisfiability to Satisfiability Modulo Theories. In *Theory and Applications of Satisfiability Testing (SAT '06)*, volume 4121 of *LNCS*, pages 1–9, 2006.

[230] Joseph Sifakis. Modeling Real-Time Systems-Challenges and Work Directions. In *Proceedings of the 1st International Workshop on Embedded Software (EMSOFT '01)*, volume 2211 of *LNCS*, pages 373–389, 2001.

[231] Joseph Sifakis. Modeling Real-Time Systems. In *Proceedings of the 25th IEEE International Real-Time Systems Symposium (RTSS '04)*, pages 5–6, 2004.

[232] Michael Sipser. *Introduction to the Theory of Computation.* Springer-Verlag, 2003.

[233] Spark Team. The SPARK Ravenscar Profile. White Paper available from http://www.praxis-his.com/sparkada/publications_tech.asp, November 2005.

[234] Cary R. Spitzer. *Digital Avionics Handbook: Elements, Software, and Functions.* CRC Press, 2nd edition, 2007.

[235] J. Springintveld, Frits Vaandrager, and P.R. D'Argenio. Testing Timed Automata. *Theoretical Computer Science*, 254(1–2), 2001.

[236] Friedhelm Stappert and Carsten Rust. Worst Case Execution Time Analysis for Petri Net Models of Embedded Systems. *Embedded Systems and Applications*, 2003.

[237] Robert Stärk, Joachim Schmid, and Egon Böger. *The Java and the Java Virutal Machine: Definition, Verification, Validation.* Springer, 2001.

[238] Steven M. Stern. An Extensible Object-Oriented Executor for the Timeliner User Interface Language. Master's thesis, Massachusetts Institute of Technology, 2005.

[239] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley, third edition, 1997.

[240] Frank van Harmelen, Manfred Aben, Fidel Ruiz, and Joke van de Plassche. Evaluating a Formal KBS Specification Language. *IEEE Expert: Intelligent Systems and Their Applications*, 11(1):56–62, 1996.

[241] Jennifer Versperman. *Essential CVS.* O'Reilly, second edition, 2006.

[242] Hans Van Vliet. *Software Engineering: Principles and Practice.* John Wiley and Sons, 2001.

[243] Norbert Völker. Thoughts on Requirements and Design of User Interfaces for Proof Assistants. In *Proceedings of the International Workshop on User Interfaces for Theorem Provers (UITP '03)*, volume 103. Electronic Notes in Computer Science, 2003.

[244] Farn Wang. Formal Verification of Timed Systems: A Survey and Perspective. *Proceedings of the IEEE*, 92(8):1283–1305, August 2004.

[245] Elaine Weyuker, Tarak Goradia, and Ashutosh Singh. Automatically Generating Test Data from a Boolean Specification. *IEEE Transactions on Software Engineering*, 20:353–363, May 1994.

[246] William M. Wilson, Linda H. Rosenberg, and Lawrence E. Hyatt. Automated Quality Analysis of Natural Language Requirement Specifications. Technical Report - NASA Goddard Space Flight Center's Software Assurance Technology Center, October 2006.

[247] Guido Wimmel and Jan Jürjens. Specification-Based Test Generation for Security-Critical Systems Using Mutations. *Proceedings of the 4th International Conference on Formal Engineering Methods (ICFEM '02)*, 2495:471–482, 2002.

[248] Jeanette M. Wing. A Specifier's Introduction to Formal Methods. *IEEE Computers*, 23(9), 1990.

[249] Kirsten Winter. Model Checking for Abstract State Machines. *Journal of Universal Computer Science*, 3(5):689–701, 1997.

[250] Niklaus Wirth. Program Development by Stepwise Refinement. *Communications of the ACM*, 14(4):221–227, April 1971.

[251] Xilinx. Virtex II Pro Data Sheets and Documentation. On-line documentation available from http://www.xilinx.com.

[252] Chunmin Yang, Barrett R. Bryant, Carol C. Burt, Rajeev R. Raje, Andrew M. Olson, and Mikhail Auguston. Formal Methods For Quality of Service Analysis In Component-Based Distributed Computing. *Journal of Integrated Design and Process Science*, 8(2):137–149, 2004.

[253] Stephen S. Yau, Robin A. Nicholl, Jefrey J.-P. Tsai, and Sying-Syang Liu. An Integrated Life-Cycle Model for Software Maintenance. *IEEE Transactions on Software Engineering*, 14(8):1128–1144, 1988.

[254] Steven J. Young. *Real Time Languages: Design and Development.* Ellis Horwood Publishers, 1982.

[255] Marvin V. Zelkowitz. Perspectives in Software Engineering. *ACM Computing Surveys*, 10(2):197–216, 1978.

[256] Hong Zhu, Patrick A. V. Hall, and John H. R. May. Software Unit Test Coverage and Adequacy. *ACM Computing Surveys*, 29:366–427, 1997.

[257] Miriam Zia, Sadaf Mustafiz, Hans Vangheluwe, and Jörg Kienzle. A Modeling and Simulation Based Approach to Dependable System Design. In *Proceedings of the 8th Intenational Conference on Model Driven Engineering Languages and Systems (MoDELS '05)*, volume 3713 of *LNCS.* Springer-Verlag, 2005.

[258] Marc Zimmerman, Kristina Lundqvist, and Nancy Leveson. Investigating the Readability of State-Based Formal Requirements Specification Languages. In *Proceedings of the 24th International Conference on Software Engineering (ICSE '02))*, 2002.