Automated Verification of Model-based Programs Under Uncertainty

by

Tazeen Mahtab

Submitted to the Department of Electrical Engineering and Computer Science

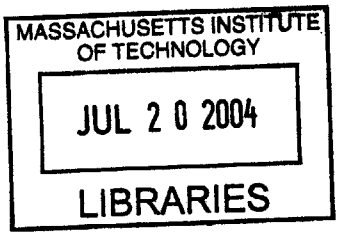
in Partial Fulfillment of the Requirements for the Degrees of

Bachelor of Science in Computer Science and Engineering

and Master of Engineering in Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology

May 20, 2004
[June 2004]



Author .....
Department of Electrical Engineering and Computer Science
May 20, 2004

Certified by .....
Gregory T. Sullivan
Thesis Supervisor

Certified by .....
Brian C. Williams
Thesis Supervisor

Accepted by .....
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

Automated Verification of Model-based Programs Under Uncertainty

by

Tazeen Mahtab

Submitted to the Department of Electrical Engineering and Computer Science

May 20, 2004

In Partial Fulfillment of the Requirements for the Degrees of
Bachelor of Science in Computer Science and Engineering
and Master of Engineering in Electrical Engineering and Computer Science

Abstract

Highly robust embedded systems have been enabled through software executives that have the ability to reason about their environment. Those that employ the *model-based autonomy* paradigm automatically diagnose and plan future actions, based on models of themselves and their environment. This includes autonomous systems that must operate in harsh and dynamic environments, like deep space. Such systems must be robust to a large space of possible failure scenarios. This large state space poses difficulties for traditional scenario-based testing, leading to a need for new approaches to verification and validation.

We propose a novel verification approach that generates an analysis of the most likely failure scenarios for a model-based program. By finding only the most likely failures, we increase the relevance and reduce the quantity of information the developer must examine. First, we provide the ability to verify a stochastic system that encodes both off-nominal and nominal scenarios. We incorporate uncertainty into the verification process by acknowledging that all such programs may fail, but in different ways, with different likelihoods. The verification process is one of finding the most likely executions that fail the specification. Second, we provide a capability for verifying executable specifications that are fault-aware. We generalize offline plant model verification to the verification of model-based programs, which consist of both a plant model that captures the physical plant's nominal and off-nominal states and a control program that specifies its desired behavior. Third, we verify these specifications through execution of the RMPL executive itself. We therefore circumvent the difficulty of formalizing the behavior of complex software executives.

We present the RMPLVerifier, a tool for verification of model-based programs written in the Reactive Model-based Programming Language (RMPL) for the Titan execution kernel. Using greedy forward-directed search, this tool finds as counterexamples to the program's goal specification the most likely executions that do not

achieve the goal within a given time bound.

Thesis Supervisor: Gregory T. Sullivan
Title: Research Scientist

Thesis Supervisor: Brian C. Williams
Title: Associate Professor of Aeronautics and Astronautics

Acknowledgments

I would like to begin by thanking my advisors, Greg Sullivan and Brian Williams, for their invaluable guidance and support in the completion of this thesis.

A huge thanks is due to Oliver Martin and Paul Elliott, for answering my innumerable questions and always being ready to help. I could not have done it without you.

Thank you to everyone in the MERS group. It was great having the opportunity to work with you. A special mention goes to my fellow Tech Square residents, Jon Kennell, I-hsiang Shu, Raj Krishnan, Judy Chen, Stanislav Funiak, and Brad Hasegawa - I value your friendship. I would also like to thank Margaret Yoon, our administrative assistant.

To Tamara Yu and Yukie Tanino, thanks for getting me through my final year.

Lastly, I wish to thank my family for their love and support.

This thesis was supported by a MOBIES grant from the Air Force Research Lab under Contract# F33615-00-C-1702.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Highly robust embedded systems have been enabled through software executives that have the ability to reason about their environment. Such systems must often operate in harsh and dynamic environments, like deep space, and therefore must be robust to a wide combination of potential failures. Those that employ the *model-based autonomy* paradigm automatically diagnose and plan future actions, based on models of themselves and their environment. *Model-based programming* is an approach to developing embedded systems that can reason about and control their hardware using corresponding software models [20]. A *model-based program* enables one to specify the desired state evolutions of the embedded system. It consists of a specification of behavior, known as a *control program*, and a representation of the physical plant's nominal and off-nominal states, known as a *plant model*; these are run on a *model-based executive*. Model-based systems have the ability to detect and respond to unanticipated failures on the fly. Therefore, they provide an increased assurance of reliability and *fault awareness*. However, such programs present a new challenge to verification. First, the large space of failure situations they handle poses difficulties for traditional scenario-based testing. Second, they are run on a complex execution algorithm. This leads to a need for new kinds of verification and validation [15].

Verification and validation (V&V) is an established methodology for ensuring the quality and reliability of software systems. By definition, verification assures that a product satisfies its requirements at a given phase in the development cycle,

and validation assures that the final product satisfies the system requirements [16]. Previous verification efforts, such as the symbolic model checking of reactive programs [3], have focused on determining the correctness of embedded programs. However, in the real world, where embedded programs control real hardware, those systems are never guaranteed to succeed; they are more or less likely to succeed. We extend model-based system verification to the verification of model-based programs under uncertainty.

We propose a novel verification approach that generates an analysis of the most likely failure scenarios for a model-based program. First, we provide the ability to verify a stochastic system that encodes both off-nominal and nominal scenarios. We incorporate uncertainty into the verification process by acknowledging that all such programs may fail, but in different ways, with different likelihoods. Second, by verifying model-based programs, we provide a capability for verifying executable specifications that are fault-aware. A model-based system is never guaranteed to function correctly, since it always has some probability of not behaving nominally. Therefore, verification of these systems is a process of finding the most likely of these failure executions, rather than simply determining if there are any. Third, we verify these specifications through execution. We therefore circumvent the difficulty of formalizing the behavior of complex software executives by invoking the actual executive components.

We present the RMPLVerifier, a tool for verification of model-based programs written in RMPL for the Titan execution kernel. The *Reactive Model-based Programming Language (RMPL)* allows developers to perform high-level reasoning on the behavior of a model. The RMPLVerifier uses greedy forward-directed search to analyze an RMPL model-based program based on a goal specification and a given time bound. It then presents, as counterexamples, the most likely executions that lead to failure i.e. non-achievement of the stated goal. We analyze the results of applying our verification approach to the Mars Entry scenario, a significant model-based program specifying the entry sequence for a lander spacecraft.

The remainder of this chapter will motivate the verification of model-based pro-

grams, relate verification to the Mars Entry scenario, and give an overview of the RMPLVerifier.

1.1 Motivation

Our verification approach provides three capabilities. We examine and motivate each of these in turn.

The first contribution of our approach is the ability to verify a stochastic system that encodes both off-nominal and nominal scenarios. In the past, one created robust embedded systems by attempting to enumerate ahead of time all possible failures the system could encounter. These systems were limited by the ability of the software development team. If a system encountered a failure that had not been predetermined by the developers, possible because of the many complex interactions between the hardware and software and the environment, it could fail to react properly. For example, the failure of the Mars Polar Lander is thought to have occurred because of unexpected leg sensor readings as it attempted to land. The software erroneously concluded from these readings that the Lander had touched down on the surface and prematurely shut off the engines, leading to the loss of the craft. New intelligent systems have been created to address this problem [20]. Rather than being preprogrammed with all failures, these systems have the ability to deduce if they are in a nominal or failure state and to respond accordingly. These systems are stochastic, as they maintain knowledge of the probability of being in a particular state at a given time [20]. We provide a verification capability for such systems. By returning information on the likelihood of the program's executions, we incorporate uncertainty into the verification process. By showing only the most likely failure executions, the RMPLVerifier helps focus the systems engineer on the most vulnerable components of a system.

Our second contribution is the ability to verify executable specifications that are fault-aware. Synchronous programming languages, such as Esterel [1], were designed for writing software to control reactive systems. A synchronous programming lan-

guage is characterized by logical concurrency, orthogonal preemption, multiform time, and determinacy, which have been shown to be necessary characteristics for reactive programming [1]. Synchronous programming seeks to provide executable specifications. An *executable specification* is a program that doubles as a specification about which one can prove properties and an executable implementation of that specification. This eliminates the need to write a specification and implementation separately [20]. Model-based programming generalizes this approach to executable specifications that are *fault-aware* - they have knowledge of the behavior of the plant under failures as well as nominal situations. A model-based program, consisting of a control program and plant model, is a fault-aware executable specification of the desired behavior of a robust embedded system. The plant model is a representation of the hardware, including the nominal and faulty states it may be in. The control program directs the actions of the executive to progress the system through a sequence of intended states. The executive uses the plant model to generate a control sequence that achieves these intended states. The verification task for a model-based program therefore has two pieces. One may verify properties of the plant model alone, or one may verify the control program, given a correct plant model. Our work focuses on the latter, while previous work has focused on the former [15]; there has also been research on the verification of the diagnostic executive [7] [12]. The control program, by its nature, has a high-level goal. For example, this could involve carrying out a navigation procedure or maintaining a sub-system in a steady-state. We enhance the control program to include this definition of success in the form of a goal specification. The results returned by verification are counterexamples to this overall specification.

Our final contribution is the ability to verify these specifications through execution. To handle all possible failure scenarios, the reactive systems that we have described must consider an exponentially large state space. To achieve tractability model-based executives consider a subset of the possible situations and solutions, by employing anytime algorithms. Due to this approximate inference, it is difficult to formalize the behavior of such systems correctly. In addition, changes to approximations made by the reactive system over time would render any formalization used

by a verifier obsolete. We therefore generate our results by running the specification on the actual software executive. Our tool verifies programs written in the Reactive Model-based Programming Language (RMPL) using the Titan executive. Titan includes both a *control sequencer* and *deductive controller*. The control sequencer generates the sequence of intended states, while the deductive controller attempts to achieve them. An RMPL model-based program can have many different executions, which depend on the observations it receives, the time for which it runs, and the mode estimation algorithm used for diagnosis. Some of these executions will achieve the program's goal, and others will not. For instance, along one execution path, a camera may fail to take a picture, resulting in an unsuccessful navigation procedure. The verification tool focuses on these unsuccessful execution paths. It explores the set of most likely executions over the specified number of program steps. It interfaces directly with the Titan executive and can thus easily accommodate updates to Titan.

We further motivate our verification approach with an example.

1.2 Example of Verification on a Model-based Program

Consider the problem of controlling a spacecraft system. A spacecraft has hundreds of different components that must interact in complex ways. At the same time, a spacecraft operates autonomously in an unpredictable environment, making it likely that there will be unexpected failures. These properties make it a good candidate for model-based autonomy. Figure 1-1 shows a Mars lander spacecraft. Figure 1-2 shows the RMPL control program [11] specifying the desired behavior of such a spacecraft during an entry scenario. The program performs a series of actions in preparation for entering the atmosphere of Mars, such as turning on the engine and letting it heat to standby, changing the kind of navigation used, and properly orienting the craft. The program operates on a model of the spacecraft, which represents both nominal and failure scenarios. Thus, an engine can be in the states firing or failed.

Figure 1-1: The Mars Polar Lander. *Courtesy NASA/JPL-Caltech.*

```
1 EntrySequence() {
2    Engine = Standby;
3    Nav = Inertial;
4     do {
5        always (Att = Entry-Orient),
6        when (Att = Entry-Orient)
7           donext (Lander = Separated)
8    } watching (Entry = Initiated)
9 }
```

Figure 1-2: The RMPL Control Program for the Mars Entry Scenario.

Since many failure scenarios are possible, a developer creating a model-based program for such a system would find it beneficial to be able to enumerate possible failure executions. The RMPLVerifier returns the most likely executions of a model-based program that do not lead to success. The tool tracks a set of most likely plant state trajectories over time, as shown in Figure 1-3. The figure shows a simplified trajectory that could be returned as a counterexample by the verifier. In this trajectory, the engine transitions from off to heating and then to a failed state. We revisit the Mars Entry example in Chapter 5.

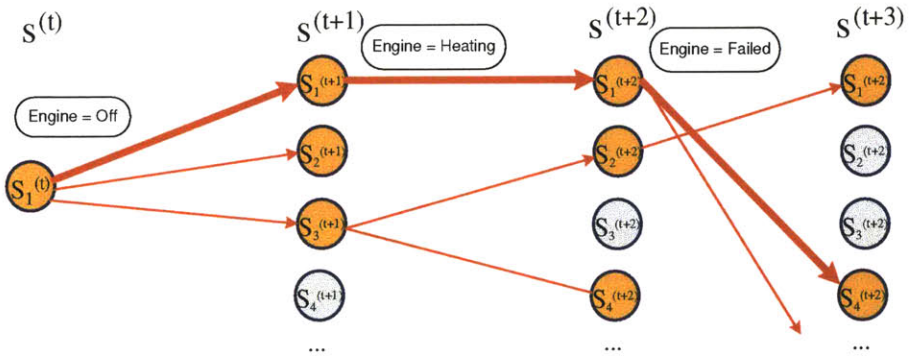


Figure 1-3: The Set of Plant State Trajectories Tracked by the RMPLVerifier.

1.3 The RMPLVerifier

This thesis presents RMPLVerifier, a development tool for RMPL model-based programs. The RMPLVerifier verifies a model-based program against a specification of success. It takes as input a model-based program with a goal specification and produces as counterexamples the k most likely failure executions of the program. The verifier generates these executions by searching the space of possible trajectories using greedy forward-directed search.

The RMPLVerifier generates trajectories using Titan and a simulator that provides observations consistent with the plant state. A *plant state trajectory* includes only states of the plant model and is tracked by the simulator. A *plant state* has a likelihood, computed from the likelihood of the previous state and the probability of transitioning to it from the previous state. A *program state* includes the states of the control program and plant estimate at a given point in the execution of a model-based program. A *program state trajectory*, which is generated by Titan, consists of a sequence of program states and represents the execution of a model-based program from the start state to a given time step for a given plant trajectory. The verifier returns a list of plant state trajectories. The likelihood of the plant state trajectories is used as the search heuristic. The search completes once the requested k number of solutions has been found for the given horizon. A list of trajectories, sorted in order of likelihood from highest to lowest, is returned as solutions to the verification query.

1.4 Thesis Layout

Chapter 2 of this thesis discusses related work on the verification of reactive systems. Chapter 3 defines the RMPL model-based program. It also gives background on Titan and its components. Chapter 4 defines the verification problem and presents the details of the verification algorithm. Chapter 5 introduces the Mars Entry scenario and works through a verification of the model-based program. Chapter 6 describes the implementation, and Chapter 7 gives results and conclusions.

Chapter 2

Related Work in Verification

Verification and validation (V & V) has been shown to improve software quality, yielding a number of benefits [16].

- It can detect errors early, reducing overall development cost and granting time for a comprehensive solution.
- It can evaluate the product against system requirements. For example, these may be properties required by the customer.
- It gives an incremental preview of performance of the product, allowing for early adjustments. For example, it may detect early on that a program runs too slowly for the desired application.
- It may indicate whether or not to proceed to the next development phase.

V & V is especially applicable to embedded systems that must operate in environments in which they must handle a broad set of failures. Traditionally, such systems have been checked for reliability through extensive manual testing, using simulations of nominal and off-nominal scenarios. However, the number of possible scenarios can be very large for highly autonomous software, limiting the effectiveness of traditional testing [2]. This chapter gives an overview of methodologies relevant to the V & V of software for embedded systems.

2.1 Model Checking

Model checking is a methodology which exhaustively explores a system's achievable states. It is a common technique for verifying digital hardware and reactive software. Given a model of a system and a property for correctness, a model checker runs through all possible executions of that system, including interleavings of concurrent threads, and reports any execution that violates the property specification. Explicit state model checking generates and explores every state [13]. Symbolic model checking efficiently manipulates compact encodings of sets of states and, therefore, may be applied to larger systems [15]. A drawback of model checking is that the system usually is converted beforehand, into the formal syntax accepted by the model checker. This is generally a tedious process carried out manually by an expert. Alternately, translators can be used; we describe one later in this chapter [15].

Model checking has been demonstrated for verifying the correctness of plant models used in model-based systems, as we see when we describe the Livingstone to SMV translator [15]. It has also proven useful for identifying bugs related to concurrency in the source code of system executives [7]. However, there are difficulties in applying model checking to verification of the executions of a model-based system, the goal of this thesis. A model-based system is difficult to translate into a formal language, a requirement for model checking. In addition, the space of executions model checking must explore is exponentially large.

We now describe applications of model checking to reactive systems.

2.1.1 Livingstone to SMV

Livingstone [21] is a model-based diagnostic system developed as part of the Remote Agent autonomous spacecraft controller [14]. Livingstone uses a plant model to describe the nominal and failure modes of each component in the system. Each mode has associated *modal constraints* that describe the behavior of the component when in that mode. A component also has transition constraints that determine when it can switch between modes. Plant models are written in the Model Programming

Language (MPL). The goal of the Livingstone to SMV project was to apply formal verification techniques to the development of autonomous controllers based on Livingstone. The focus was on diagnosing errors in plant models. Properties for correct models included reachability of component modes in the plant model and consistency and completeness of mode transitions. Transitions are consistent when only one of a mode's outgoing transitions to other modes is enabled at a time; two transitions cannot be simultaneously enabled. Transitions are complete when at least one transition constraint is always fulfilled.

The Livingstone to SMV translator [15] converts the plant models used by Livingstone into specifications that can be verified with the Symbolic Model Verifier (SMV) [3], a model checker from Carnegie Mellon University. The specification for correctness is expressed in terms of properties in the temporal logic CTL (Computation Tree Logic) or, alternately, using pre-defined specification patterns in MPL, Livingstone's modeling language. The translator converts both the plant model and the specification from MPL to SMV's formal language, and then converts any diagnostic output from SMV back to MPL. The translator thus saves the developer from tedious manual translation. The Livingstone to SMV translator has been used to verify properties of the Livingstone model of the Deep Space 1 spacecraft used within the Remote Agent experiment.

The Livingstone to SMV project only addressed the verification of the plant models of model-based systems. This thesis seeks to verify an entire model-based program, including both control program and plant model.

2.1.2 Remote Agent

A research effort [7] was conducted that applied formal verification methods to the RA Executive [14], one of the three subsystems of the Remote Agent autonomous spacecraft controller, demonstrated in flight on the Deep Space 1 mission. The Executive provides operating-system level capabilities for goal-directed software. Two separate efforts applied the SPIN model checker [9] to this piece of software. The focus was to find errors in the source code of the executive.

The first effort [7] occurred during the development process and early on found five concurrency errors that would not have been found through testing. The lisp source code of the Executive was abstracted and translated by hand to the PROMELA language used by SPIN, a model checker for analyzing the correctness of finite state concurrent systems. The model checker then examined it for properties such as the liveness of concurrent tasks. The significant manual labor required motivated the creation of the Java PathFinder tool [18], a translator from Java to PROMELA. As mentioned in the next section, this tool was also later used during a V&V survey on the Rover Executive. The second effort [7] occurred due to an in-flight deadlock in a sibling subsystem to the one verified. The RA developers provided the researchers the code for the Executive without identifying where the bug was located. Researchers conducted a separate "clean-room" verification and found the concurrency error in a total process of two days; most of the time was spent in understanding and modeling the code. The project demonstrated that formal verification can find concurrency errors that occur in actual flight. It also produced tools that increased the ease with which V&V could be incorporated into the software development cycle. As one of the more successful applications of formal methods, it gave an impetus for more research in the verification of such systems.

The authors presented one of the successful applications of verification to software that was part of a reactive system. However, they focused exclusively on source code verification, the realm of traditional V&V. In this thesis we look instead at verification of the higher-level behavior of the executive, as part of the model-based system that includes the model-based program.

2.1.3 Rover Executive V & V Study

We now give an overview of a study conducted at NASA Ames for the purpose of determining the maturity of different verification and validation techniques, including model checking, on a representative example of NASA flight software [2]. The study had two goals. The first was to evaluate the relative strengths and weaknesses of traditional and advanced V&V approaches and tools on autonomy software. The

second was to determine if current tools show that advanced verification techniques are a significant improvement over traditional ones.

The study consisted of a controlled experiment where three V&V technologies (static analysis, runtime analysis and model checking) were compared to traditional testing regarding their ability to find seeded errors in the source code of a Rover executive. The Rover Executive [19] was a software prototype written in C++ by researchers at NASA Ames. It commanded a rover through the use of high-level plans. Groups of two were assigned to each methodology and given the task of finding bugs. Static analysis was conducted with the PolySpace Verifier [17], runtime analysis with the DBRover [4] and Java PathExplorer [8], and model checking with the Java PathFinder [18].

The significance of this study is that it was the first to experimentally evaluate and compare formal methods-based tools to testing on a realistic research prototype of embedded software. While the authors did not draw any strong conclusions, the study did confirm that advanced tools can outperform manual testing when trying to locate concurrency errors. They also drew a number of insights. They found runtime analysis and monitoring to be very successful in uncovering the seeded bugs. Runtime analysis methods detect the correctness of a program by collecting data from the execution of the program and then analyzing it. Runtime monitoring validates the correctness of a single execution online against a set of formally state requirements. However, as both of these techniques require the code to be executed, their effectiveness was limited to how exhaustively the test-inputs covered the input space. The authors also found model checking to be good at systematic analysis; for example, it could systematically cover all input plans up to a specific size for the rover.

This study gives basis to our claim that there is a real benefit to using automatic verification tools on reactive systems, when compared to the results obtained from manual testing.

2.2 Executable Specifications

An *executable specification* is a program that doubles as a specification about which one can prove properties and an executable implementation of that specification. This eliminates the need to write a specification and implementation separately [20]. We look at Esterel, a language that creates executable specifications.

Esterel [1] is a synchronous programming language. Synchronous languages were designed to program *reactive systems*, systems that have a reactive program as their main component. *Reactive programs* are programs that maintain an interaction with their environment, reacting to inputs received from the environment by sending outputs to it. For example, an operating system driver is a reactive program embedded in a larger system. Reactive programs are composed of three layers. An interface controls input reception and output production. A *reactive kernel* decides what computation and outputs to generate in response to the inputs. A data handler performs the computations requested by the reactive kernel. Esterel is not a full programming language but a language for defining reactive kernels, which can then be generated as code in another language. It is analogous to the grammar from which a parser generator produces a parser. Once generated, the reactive kernel can be embedded in a larger program that implements the interface and data handling layers.

Esterel programs are executable specifications. The main theorem of Esterel is that the behavioral and execution semantics are equivalent. The behavioral semantics give a mathematical definition of the semantics of the language. The execution semantics define the actions of the underlying execution machine. Esterel is not a high-level specification language that requires the developer to rewrite the program in an implementation language once the specification is complete. Rather, the specification and implementation are one and the same. Once written, a program can be efficiently compiled to object code.

RMPL is another synchronous language that builds on the ideas of Esterel. RMPL model-based programs are executable specifications that are fault-aware - they know the plant's behavior under both failure and nominal scenarios. This thesis performs

verification on RMPL executable specifications by executing them using the Titan engine.

2.3 Simulation-based Verification

Livingstone PathFinder (LPF) [12] uses a new verification approach based on simulation. It seeks to find diagnosis errors in the Livingstone engine, instances when the engine inaccurately diagnoses the current state of the system. LPF applies state space exploration algorithms to a testbed, consisting of the Livingstone [21] engine embedded in a simulated environment. The simulator is generally also a Livingstone engine. The tool runs on a Livingstone plant model and a low-level scenario of commands and injected faults. It has the ability to report error conditions during execution of the scenario on the model. It can report when the engine fails to find any consistent candidate for the current state. It can report a discrepancy between the component modes of the simulator and diagnoser. For example, if the plant model has a valve component which the simulator believes to be in the **open** mode and the diagnoser believes to be in the **closed** mode, the tool reports that there has been a misdiagnosis.

LPF runs on a set of different executions defined from the permutation of the events given by the scenario. The events of the scenario are the commands and faults; the faults are interleaved in the sequence of commands. LPF explores the tree of executions given by the scenario using search, saving and restoring intermediate execution points. The user may specify the kind of search, where the options are depth-first search with backtracking, breadth-first search, or best-first search, with a heuristic that favors situations where fewer candidates for the estimated state were found and therefore a misdiagnosis is more likely to occur. The user may also specify whether to report one or all errors.

The focus of LPF is primarily on reporting diagnosis errors in the engine. A disadvantage of the tool noted by the authors was that it generated a large amount of output and could therefore benefit from improvement in the post-treatment and display of results. Currently Livingstone PathFinder only uses plant models and the

deductive controller. However, a new version, called Titan PathFinder, is currently in development. This version seeks to use Titan as the diagnostic engine, including both the deductive controller and the control sequencer; the task is still to find misdiagnoses.

LPF employs a similar strategy to the RMPLVerifier, both tools using search to explore executions of a model-based system. However, LPF focuses on finding misdiagnoses. Our tool focuses on the model-based program, finding executions of the program that do not achieve success, along with their likelihoods.

2.4 Summary

The verification research on reactive systems that we have examined has focused primarily on verification of those systems, while ignoring the failure behavior of the plant being controlled. In addition, in the area of model-based autonomy, verification has focused primarily on the detection of errors in the plant model or the implementation of the diagnostic engine. Our focus is to develop verification approaches for an entire model-based program, which includes both a plant model and a control program. In addition, these past efforts have focused on determining the correctness of model-based systems. We instead account for uncertainty by determining how likely a program is to succeed or fail.

Chapter 3

The RMPL Model-based Program

Languages for embedded systems have been developed to simplify the task of controlling devices with many different components. These languages enable one to write programs that interact with the hardware by reading sensor values and by sending commands to actuators. In programming languages like Esterel [1] and Statecharts [6], the programmer is responsible for mapping the intended state of the system to the variables holding sensor and actuator values. The complex interactions between hardware components and the large number of failure scenarios possible make this mapping error-prone. The *model-based programming* paradigm [20] was designed to address this problem. A model-based programming language interacts directly with the state of the plant. Programmers are given the ability to define and control hidden state variables; by manipulating a state variable, one indirectly interacts with observable and control variables corresponding to sensors and actuators respectively.

The RMPLVerifier, the central focus of this thesis, performs verification on an RMPL model-based program. In this chapter, we describe the Reactive Model-based Programming Language. Section 3.2 defines a model-based program. Section 3.1 introduces the RMPL language with the help of an example. Section 3.3 describes Titan, RMPL's execution kernel, and includes a detailed discussion of the Mode Estimation component of Titan. The details of Mode Estimation are particularly important background for verification, as the prediction step of Mode Estimation forms the core of the verifier's trajectory generation algorithm.

3.1 The Reactive Model-based Programming Language

The *Reactive Model-based Programming Language (RMPL)* [20] allows developers to perform high-level reasoning on the behavior of a model. It is an object-oriented language that uses propositional state logic as its underlying constraint system. RMPL provides primitive constructs for conditional branching, preemption, iteration, and concurrent and sequential composition. We highlight the key design features of the language with the aid of an example.

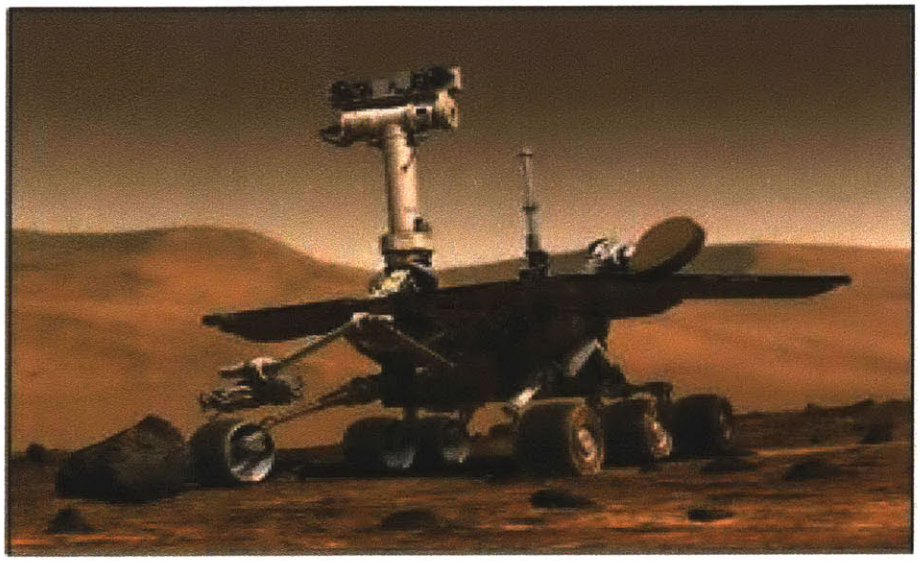


Figure 3-1: The Mars Exploration Rover. *Courtesy NASA/JPL-Caltech.*

A rover's mission is to drive to a final destination (Figure 3-1). In order to reach it, it drives to a set of intermediate waypoints. It stops at each to take an image with its hazard camera, to be used by its on-board hazard avoidance algorithm. We can translate this scenario into a model-based program in the following manner.

Figure 3-2 shows the RMPL control program. It is expressed in a style similar to traditional software programs. In the main procedure, **DriveToTarget()**, the rover initially takes a picture with the **TakeHazardPicture()** subprocedure and drives to the next waypoint with the **DriveToNextWaypoint()** subprocedure. When it has

```
1   DriveToTarget() {
2     do {
3         {
4           TakeHazardPicture();
5           DriveToNextWaypoint();
6         } ,
7          when (RoverWaypoint = Reached)  donext {
8             TakeHazardPicture();
9             DriveToNextWaypoint();
10        }
11    } watching (RoverTargetPosition = Reached or HazcamHealth = Unhealthy)
12  }
13
14  TakeHazardPicture () {
15  ;; In full version, we include the stereo image, terrain map and path
16  ;; generation.
17      RoverMotion = Park;
18      HazcamImage = Valid;
19   }
20
21  DriveToNextWaypoint () {
22  ;; In full version, we maintain the Rover in a driving state until the
23  ;; Waypoint is reached. Actual control of the driving is assumed to occur
24  ;; at some lower control layer.
25     do {
26          always RoverMotion = Arc
27    } watching (RoverWaypoint = Reached)
28  }
```

Figure 3-2: A Control Program for the Rover-Hazcam Scenario.

reached this waypoint, it repeats its actions. In this way, the rover drives to each intermediate waypoint, stopping to take a hazard picture at each before proceeding. If, at any point, it reaches its target position or detects that the state of the camera is unhealthy, the procedure terminates.

This example highlights several important properties of a control program. First, the code is written in terms of state assignments. Assignments may be used as execution conditions. The assignment **RoverTargetPosition = Reached** is an execution condition that depends on the plant's hidden state variable **RoverTargetPosition** (Line 11). State assignments may also be used as assertions. The assignment **Rover-**

Motion = Park (Line 17) in the **TakeHazardPicture()** procedure directs the system to make that assertion true once execution reaches that point in the program. These hidden state assignments are not directly observable or controllable. They provide a layer of abstraction that makes it easier for the developer to write a reactive program. This specification is far simpler than a program that must turn on motors and switches directly. Second, RMPL allows both parallel execution, denoted by a comma, (Line 6) and sequential execution, denoted by a semicolon (Line 5). RMPL also allows both conditional execution and iteration. The **when ... donext** construct on Line 7 is an example of both; the actions inside the construct repeat until the condition, **RoverWaypoint = Reached**, is satisfied. Finally, the language allows preemption. The procedure **DriveToTarget()** terminates if either **RoverTargetPosition = Reached** or **HazcamHealth = Unhealthy** becomes true (Line 11).

The plant model is a representation of the rover's behavior, and is used by the Titan executive to achieve the control program. It is a system composed of several subcomponents, each illustrated by a rectangle in Figure 3-3. These components interact with the hardware through sensor and actuator values. They may also interact with each other. The figure uses arrows to show incoming observations and outgoing commands, as well as the flow of internal information between components. The RoverMotion component describes the movement of the rover; it can command it to drive or stop. The RoverWaypoint and RoverTargetPosition components check to see if the rover has reached a waypoint and its target destination respectively; they receive position information as observations. The HazcamOperation, HazcamHealth, and HazcamImage components represent the Hazcam camera of the rover. HazcamOperation can command the camera to turn on or off or to take a picture. It can observe whether the power is on or off and whether the camera is ready. HazcamImage can observe whether the image recorded is valid; it depends on input from HazcamOperation. HazcamHealth decides whether the camera is in healthy condition, depending on inputs from HazcamOperation and HazcamImage. The plant model given for this example scenario is still somewhat high-level. A model to be used

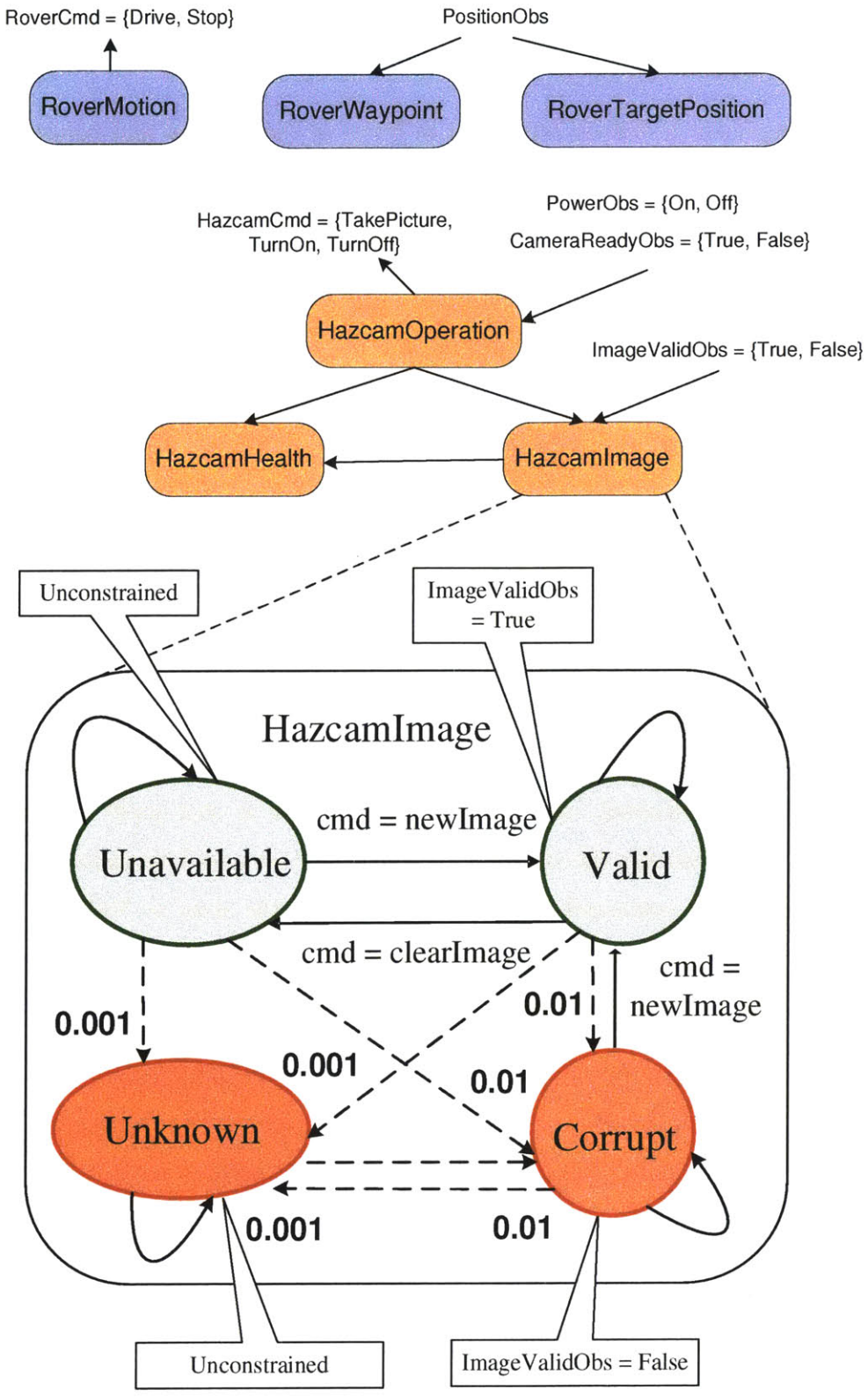


Figure 3-3: The Plant Model for the Rover-Hazcam Scenario.

in a real scenario would contain many more low-level components that describe the hardware more accurately.

Each component is defined separately in terms of the nominal and failure states it may occupy. Since components may be defined independently, it is simple to build a plant model in a modular manner by reusing components. A component's state depends on the actual observations received and commands issued to the hardware. We look in detail at the component definition of HazcamImage, the graphical representation of which is shown in Figure 3-3. HazcamImage represents the hidden state of the camera image. It has the nominal modes **Valid** and **Unavailable** and the failure modes **Corrupt** and **Unknown**, corresponding to the states that it may occupy. It observes the validity of the image from the plant as sensor readings **True** or **False**. When the image is **Valid**, we expect the observed output to be **True**, and when the image is **Corrupt**, we expect it to be **False**. Each mode in the figure is labeled with a modal constraint that expresses these conditions. For example, mode **Valid** has **ImageValidObs** = **True**. The **Unknown** mode is unconstrained, since nothing is known about the conditions of an unanticipated failure. The **Unavailable** mode is also unconstrained, since no image is available at that point for observation. The figure illustrates the commanded transitions between **Unavailable**, **Valid** and **Corrupt**. The command **newImage** transitions the state to **Valid**, and the command **clearImage** transitions it to **Unavailable**. This particular model defines relationships such that HazcamImage's commands are variables dependent on the output of HazcamOperation. However, a component may command the plant directly in the same manner. One may also take an un-commanded transition to the fault modes **Corrupt** and **Unknown** from any other mode. This indicates that the image may fail at any time. Self-transitions indicate that the state of the image has not changed between time steps. Each transition between states has a probability; only the fault transitions are labeled in the figure. For instance, upon issuing the command **newImage** when the image is unavailable, one has probability 0.001 of transitioning to **Unknown**, probability 0.01 of transitioning to **Corrupt**, and probability 0.989 of transitioning successfully to **Valid**.

As we have seen, the control program combined with the plant model compose the complete RMPL model-based program. The RMPL compiler efficiently compiles the components of a model-based program into representations of finite automata. A plant model is represented by a *Concurrent Constraint Automaton (CCA)* [20]. A CCA is a composition of concurrently operating constraint automata, along with the interconnections between component automata and with the environment. A control program is represented by a *Hierarchical Constraint Automaton (HCA)*, a variant of hierarchical automata [20]. The plant model and control program are defined in the next section.

3.2 The Model-based Program

A model-based program enables one to specify the desired state evolutions of the embedded system. It consists of a *control program* and a *plant model.* The plant model is a representation of the hardware, defining its nominal behavior and its behavior during common failures. The control program specifies the system behavior in terms of the model. The verifier returns plant state trajectories that are expressed using the variables of the plant model.

The plant model defines state variables in terms of control and observable variables, and the control program sets these state variables. We give the formal definitions of each below. The plant model is a partially observable Markov decision process $P = \langle \Pi, \Sigma, T, P_\Theta, P_T, P_O, R \rangle$, where:

- Π is a set of finite-domain variables, divided into *state variables* Π^s, *control variables* Π^c, and *observable variables* Π^o. A *state* of the plant is defined as an assignment to Π^s. Control and observable variables correspond respectively to commands issued to and observations received from the actual plant.

- Σ is the set of all *feasible* full assignments over Π. An assignment is feasible if it is allowed by the constraints inherent to the hardware. The set Σ_s, the projection of Σ on variables in Π^s, is the set of all feasible states.

- T is a finite set of transitions. A transition $\tau \in T$ is a function $\tau : \Sigma \rightarrow \Sigma_s$ that maps feasible full assignments over Π to states.
- $P_\Theta(s_0)$ is the probability that the plant has initial state s_0.
- $P_T(\tau)$ is the probability associated with transition function τ.
- $P_O(s_i, o_j)$ is the probability of observing o_j in state s_i.
- $R(s_i)$ is the reward for being in state s_i.

The control program is a deterministic automaton $CP = \langle \Sigma_{cp}, \Theta_{cp}, \tau_{cp}, g_{cp}, \Sigma_s \rangle$, where:

- Σ_{cp} is the set of *program locations*. A program location is defined as the state of the control program.
- Θ_{cp} is the initial location.
- τ_{cp} is the transition function between locations, conditioned on plant states of P. In other words, $\tau_{cp} : \Sigma_{cp} \times \Sigma_s \rightarrow \Sigma_c p$.
- $g_{cp}(l)$ is the *configuration goal* of location l. Each program location has a configuration goal, which is the set of legal plant goal states associated with that location.
- Σ_s is the set of all feasible states of the plant model.

One executes a model-based program by generating a control sequence that moves the plant to the states, known as configuration goals, specified by the program. A *model-based executive* executes a program by continuously generating these configuration goals based on the plant model and the current plant state. A configuration goal is translated to commands that are sent to the plant. The executive also continually estimates the current plant state based on the plant model and sensor data. The executive uses this information to determine if goals were successfully achieved and to diagnose failures.

3.3 Titan

Titan [20] is the model-based executive for RMPL. The architecture for Titan is shown in Figure 3-4. Titan has two main components, the *control sequencer* and the *deductive controller*. The control sequencer generates the configuration goals, based on knowledge of the state of the control program and the estimated state of the plant model. The deductive controller has two subcomponents, mode estimation and mode reconfiguration. *Mode Estimation (ME)* generates an estimate of the plant's most likely current state, based on observations received from the plant and commands issued to it. *Mode Reconfiguration (MR)* sends commands to the plant that progress the plant through the states necessary to achieve the configuration goals.

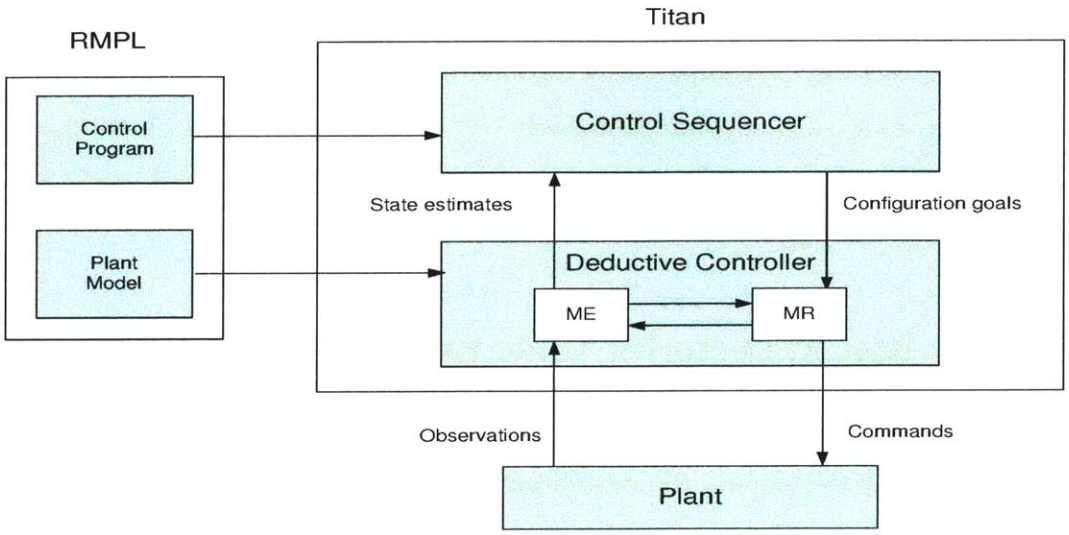


Figure 3-4: The Architecture of the Titan Model-based Executive.

3.3.1 Mode Estimation

A *plant state trajectory* is a sequence of feasible states. The space of possible state trajectories can be visualized through a Trellis diagram, shown in Figure 3-5(a). The *Trellis diagram* enumerates all possible states at each time step and all transitions between states at adjacent times. Mode Estimation (ME) is an online algorithm for

tracking the most likely states through this diagram that are consistent with the plant model, the observations and the commands. ME is an instance of Hidden Markov Model belief state update. Belief state update associates a probability to each state in the Trellis diagram. ME selects the tracked state with the highest probability as the most likely state estimate. Belief state update computes the current belief state, the probability of being in each state s_j at time $t+1$, with the following equations:

$$\sigma^{(\bullet t+1)}[s_j] = \sum_{i=1}^{n} \sigma^{(t\bullet)}[s_i]\mathbf{P}_{\mathbb{T}}(\sigma_i, s_j)$$

$$\sigma^{(t+1\bullet)}[s_j] = \sigma^{(\bullet t+1)}[s_j]\frac{\mathbf{P}_{\mathbb{O}}(s_j, o_k)}{\sum_{i=1}^{n} \sigma^{(\bullet t+1)}[s_i]\mathbf{P}_{\mathbb{O}}(s_i, o_k)}$$

$\mathbf{P}_{\mathbb{T}}(\sigma_i, s_j)$ is defined as the probability that the plant P transitions from full assignment σ_i to state s_j, computed as the sum of $\mathbf{P}_\tau$ over all transition functions τ that map σ_i to s_j. The a priori probability $\sigma^{(\bullet t+1)}[s_j]$ is conditioned on all observations up to $o^{(t)}$. The posteriori probability $\sigma^{(t+1\bullet)}[s_j]$ is also conditioned on the latest observation $o^{(t+1)} = o_k$.

3.3.2 k Best Trajectories Mode Estimation

Ideally we would like mode estimation to track all possible states of the system for all time in order to compute the most accurate estimate of the current state. However, this is very costly in practice, so ME uses approximate belief state update [10] instead. Rather than tracking the true belief state, ME computes the set of states reachable by the most likely transitions, given the latest commands and observations (Figure 3-5(b)). This approach has the limitation that a low-probability trajectory that is pruned may become very likely at a later time, after considering additional information. Since ME no longer tracks the true belief state, we cannot guarantee that the true current state is included in the set of tracked current states. An additional limitation of this approach is that it does not add the posterior probabilities for multiple separate trajectories that lead to the same state; instead, only the most likely of these trajectories is chosen. Therefore, the resulting probability of the target

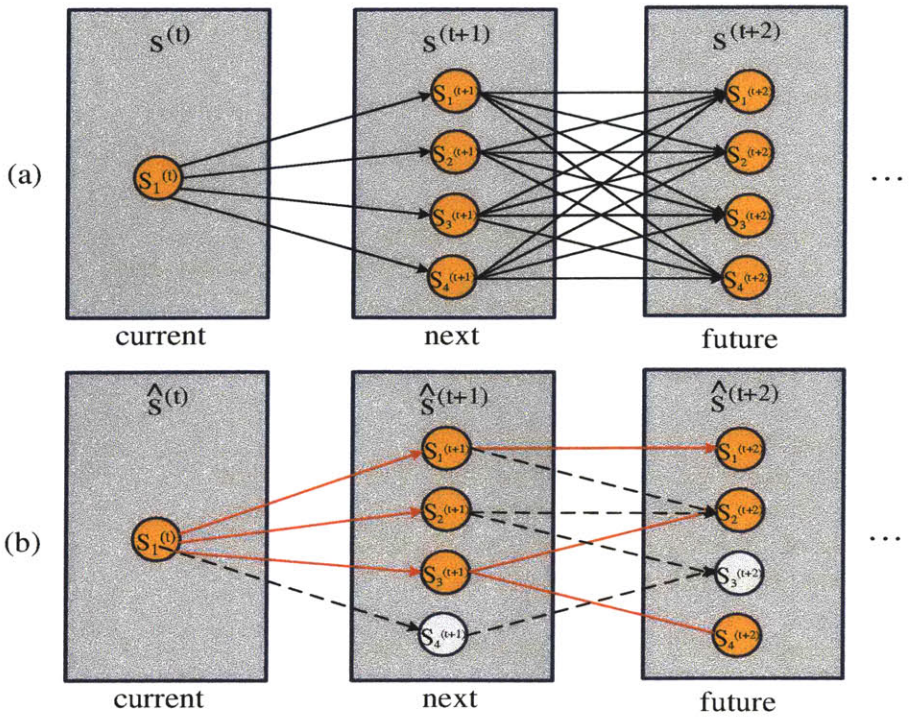


Figure 3-5: (a) A Trellis diagram showing all possible state trajectories. (b) k Best Trajectories Mode Estimation tracks only the most likely states.

state is an underestimate of the true probability of being in that state.

Figure 3-6 gives the pseudocode for the k Most Likely Trajectories algorithm [20] currently used by Titan's Mode Estimation. We begin with the set of current most likely trajectories, as well as the new command and observation received by Mode Estimation from Mode Reconfiguration and the plant respectively (Lines 1-2). A trajectory is a sequence of estimated states. For each current trajectory, we first find the most likely transition to the next state, enabled by the model and this new information (Line 8). We create a new trajectory by appending this state to the end of the current trajectory and add it to a priority queue, ordered by the trajectory probability (Line 10-11). Next, while the priority queue is not empty, we get the most likely trajectory in it and add it to the list of next trajectories (Line 14-15). If we have found k trajectories, we exit at this point (Line 16-17). If not, we find the current trajectory that originated it; this is the trajectory after its last target state

```
1  FindKMostLikelyTrajectories ( Model, CurrentTrajectories, Command,
2    Observation, NumSolns )
3  returns NextTrajectories
4
5  let NextTrajectories = {}
6  let PriorityQueue = {}
7  foreach CurrentT in CurrentTrajectories
8      compute the most likely transition from CurrentT's current state,
9        enabled by Command, Observation and Model
10     let NextT = CurrentT + target state of enabled transition
11     insert NextT into PriorityQueue
12 endfor
13 while PriorityQueue is non-empty
14    let T = pop most likely trajectory from PriorityQueue
15    insert T into NextTrajectories
16    if ( size of NextTrajectories == NumSolns )
17      return NextTrajectories
18    endif
19    let OrigCurrentT = T - last state of trajectory T
20    compute the next most likely transition from OrigCurrentT's current
21      state, enabled by Command, Observation and Model
22    let NextT = OrigCurrentT + target state of enabled transition
23    insert NextT into PriorityQueue
24 endwhile
25 return NextTrajectories
```

Figure 3-6: The k Most Likely Trajectories Algorithm.

has been removed (Line 19). We create another new trajectory, based on the next most likely enabled transition (Line 20-22). We add it to the priority queue (Line 23). This ensures that the final set of k next trajectories is the most likely, based on the possible trajectories from *all* currently tracked trajectories. If more than k trajectories are possible, these are pruned automatically as they are never inserted into the list of next trajectories.

Chapter 4

The Verification Algorithm

The RMPLVerifier is a tool for offline verification of RMPL model-based programs - both plant models and control programs - for the Titan executive. RMPLVerifier searches the space of state trajectories of the model-based program using greedy, forward-directed, best-first search. It attempts to return the k most likely program failure trajectories for a bounded time period.

In this chapter, we describe the verification problem and algorithm. Section 4.1 defines the problem we wish to solve. Section 4.2 gives an overview of the verification process. Section 4.3 gives an overview of the algorithm. Section 4.4 gives the top-level pseudocode of the verifier. Section 4.5 gives the pseudocode for the simulator component and describes the modified k Most Likely Trajectories algorithm it uses to track the most likely plant trajectories.

4.1 Problem Statement

We seek to answer the verification question "What are the k most likely plant trajectories that do not achieve a given goal within N time steps, given the control program, plant model, and starting configuration of an RMPL model-based program?"

Successful execution of a model-based program can be defined as achievement of its stated purpose, whether this is taking a picture with a camera or controlling the navigation of a satellite. Conversely, failure can be defined as not achieving this

goal. The aim of our verification approach is to find the different ways that a model-based program can fail to achieve its goal. However, a program can interact with its environment in many ways, leading to many different execution paths. We improve the relevance of our results by returning the most likely executions, along with their likelihoods. Finally, we consider only trajectories that have not achieved the goal by a finite time, as they may be impossible to determine for an infinite time bound.

4.2 The Verification Process

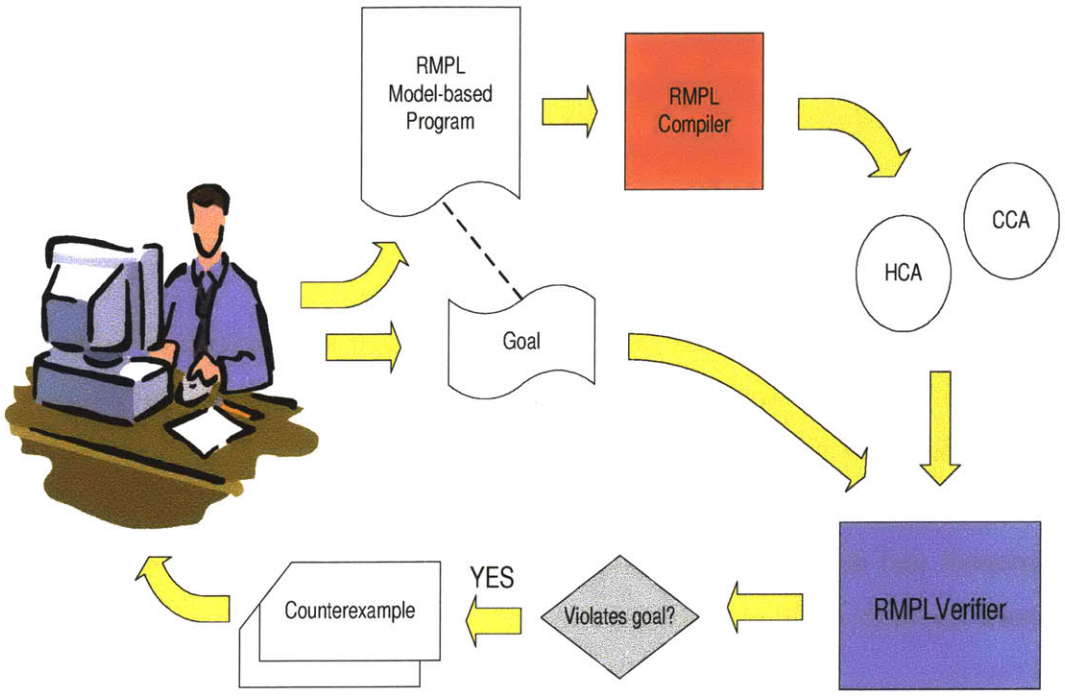


Figure 4-1: The Verification Process.

The verification process can be described by the loop depicted in Figure 4-1. The developer produces an RMPL model-based program, consisting of a control program and plant model. The control program is enhanced to include a specification of the program's high-level goal; for example, the successful firing of an engine could be expressed as the goal **(Engine = On)**.. The control program and plant model are then

compiled into Hierarchical Constraint Automaton (HCA) and Concurrent Constraint Automaton (CCA) formats, respectively, and provided as input to RMPLVerifier. The verifier uses greedy forward-directed search to discover those plant trajectories, and corresponding program executions, that do not achieve the goal within the given horizon. The verifier returns the k most likely of these. The collection of most likely program failure trajectories is then presented to the developer, who may wish to modify the model or control program based on the verification results. Each extension of a trajectory involves selecting a nominal or failure transition of the plant, and then executing the model-based program one step in response. The RMPLVerifier calls on the Titan executive to run this execution of the program.

4.3 Overview of the Algorithm

Figure 4-2 gives an overview of the action of the RMPLVerifier. It has two main components, the Titan executive and the plant simulator. The Titan executive is the same software used to control the system at runtime; it consists of the control sequencer and deductive controller. It runs on the control program and plant model. The plant simulator tracks the set of k most likely trajectories at each time step, using the model and goal specification. The simulator and Titan interact in a loop. The simulator receives commands for the current time step from the executive and returns observations consistent with the next estimated state. At the end of a given time horizon, the simulator outputs the set of most likely plant trajectories at that time step that fail to achieve the program goal. This list, sorted in order of likelihood, is returned as counterexamples to the verification query.

A *plant state trajectory* includes only states of the plant model. A *plant state* has a likelihood, computed from the likelihood of the previous state and the probability of transitioning to it from the previous state. A *program state* includes the states of the control program and plant estimate at a given point in the execution of a model-based program. A *program state trajectory* consists of a sequence of program states and represents the execution of a model-based program from the start state to

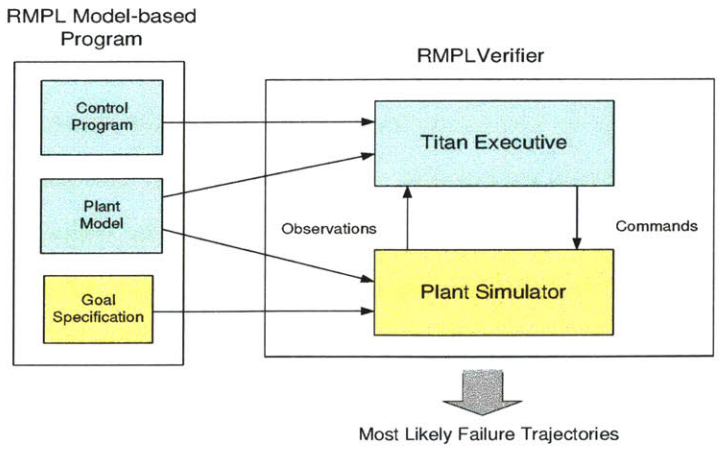


Figure 4-2: A High-level View of the Algorithm.

a given time step for a given plant trajectory. The verifier returns a list of plant state trajectories.

4.4 Top-level Pseudocode

We first describe the top-level pseudocode for the verifier, shown in Fig 4-3.

We can think of the Simulator in abstract terms as tracking a set of plant state trajectories selected from the Trellis diagram of possible trajectories. This set of plant trajectories at first contains only the initial plant state. Titan, on the other hand, can be thought of abstractly as generating a control program trajectory. The task of the verifier is to select amongst choice points in the trellis diagram, so that the simulator and Titan most likely fail to reach the goal.

In the beginning we initialize an instance of the simulator (Line 4). We next ask the simulator for the observation entailed by the initial plant state (Line 5-6). We add this initial observation to the list of observations to be passed to Titan (Line 7).

We now begin the first iteration of the verifier (Lines 9-16). We get an instance of Titan that corresponds to the simulator plant trajectory (Line 11). Since we are just starting, this will be a new Titan instance. We give our initial observation from the simulator to Titan (Line 12), as shown in Figure 4-4(a). Mode Estimation

```
1   Verify ( ControlProgram, Model, GoalSpec, InitialState, Horizon,
2     NumSolns ) returns Trajectories
3
4   let Simulator = new Simulator( Model, GoalSpec, InitialState, NumSolns )
5   let InitObservation = compute assignment to observables entailed by Simulator's
6      initial state and Model
7   let Observations = { InitObservation }
8   let Commands = { }
9   for ( let TimeStep = 0; TimeStep < Horizon; TimeStep++ )
10    for ( let i = 0; i < Observations.size(); i++)
11       let Titan = get Titan that corresponds to plant trajectory i
12       let Command = Titan.Step( ControlProgram, Model, Observations[ i] )
13       insert Command into Commands
14    endfor
15    Observations = Simulator.Step ( Commands )
16  endfor
17  return Trajectories from Simulator
```

Figure 4-3: The Top-level Pseudocode of the Verification Algorithm.

takes the observation and calculates an estimate of its initial state. The control sequencer generates the next configuration goal, based on this current state and the control program. Mode Reconfiguration issues a command for the plant based on this configuration goal, as shown in Figure 4-4(b). This is added to the list of commands (Line 13). At this point, Titan's program trajectory is composed of just one state, consisting of the states of the control program and plant estimate, originating from the initial observation. The simulator receives the new command from Titan (Line 15) and generates a new set of k observations from its plant trajectories, as shown in Figure 4-4(c).

We begin the second iteration. For each new observation, Titan is called for another step (Lines 10-14). The instance of Titan that we use must correspond to the plant trajectory for the observation (Line 11). In other words, the observation is output from a simulator plant trajectory, which generated a program trajectory using Titan in the previous iteration of the verifier. During the step, Titan's Mode Estimation learns of the new command from MR, as well as the new observation received from the simulator, and calculates the next estimated state from the current

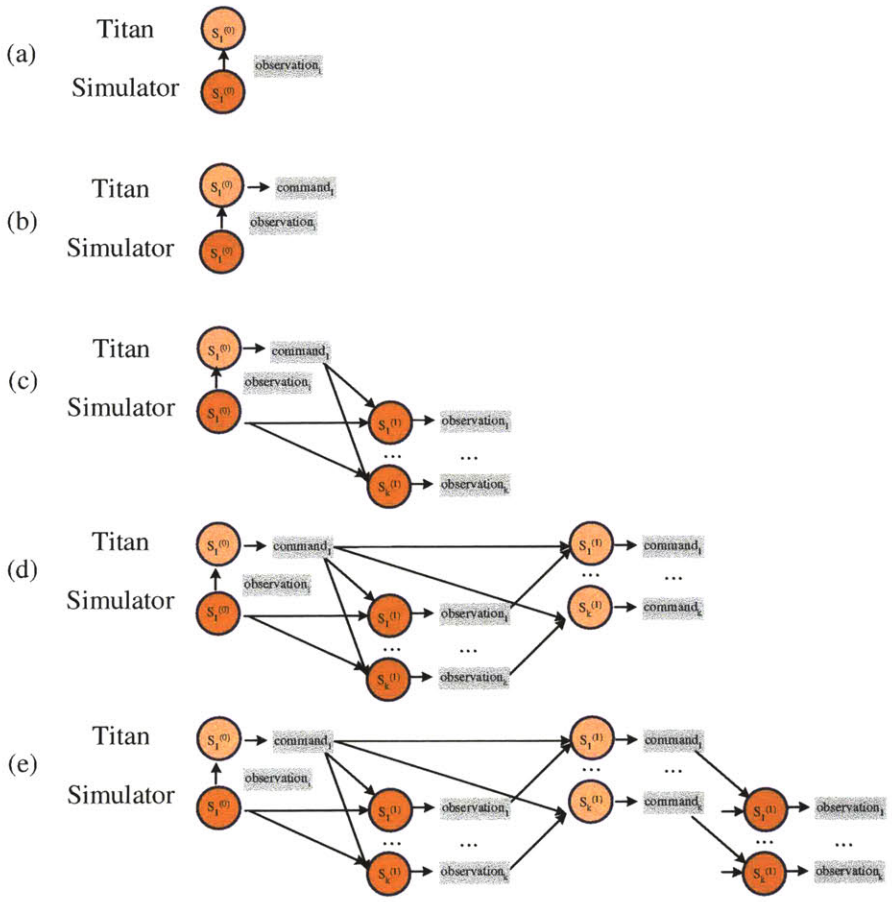


Figure 4-4: The Beginning of the Verification Algorithm. (a) The Simulator issues an observation based on the initial state. (b) Titan estimates the state based on the observation and issues a command. (c) The Simulator issues observations consistent with the next states. (d) Titan issues a new set of commands based on the new observations. (e) The Simulator again issues observations consistent with the next states.

state (Figure 4-4(d)). At this point, we have used Titan to generate k program state trajectories, obtained by extending the initial program trajectory with the different observations. We get a list of k new commands and pass them to the simulator (Line 15). The simulator ensures that each command is given to its originating plant state trajectory. This is the plant trajectory that generated the observation that in turn generated the command. The plant trajectories are extended once more, and the new observations are generated, as shown in Figure 4-4(e).

In this manner, the cycle repeats until the time horizon has been exceeded. The RMPLVerifier returns the set of most likely plant trajectories from the Simulator at the end of the last time step. Figure 4-5 illustrates the steady-state relationship between Simulator and Titan trajectories. Titan's MR issues a command. The Simulator receives that command and issues the new observation based on the next plant state. Titan's ME receives the observation as well as the command and computes its next state estimate.

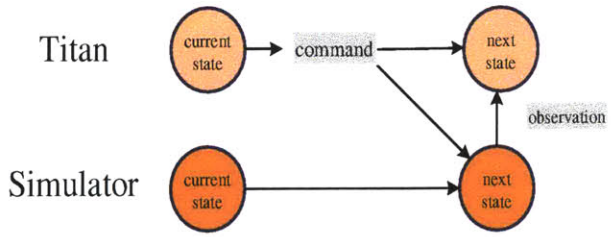


Figure 4-5: Relationship between a Titan trajectory and a Simulator trajectory.

We look into the operation of the Simulator in detail in the next section. For now, it is sufficient to know that the Simulator extends its initial trajectory based on the command by the next most likely states to a set of most likely trajectories. For each next state, the Simulator outputs an observation entailed by it. Therefore, if there are k next states, there are k corresponding observations (Figure 4-4(c)).

We assume that a state uniquely determines an observation. The next state determined by Mode Estimation based on this observation and the command may or may not be the same as the plant state. Whether or not ME correctly estimates the

plant state depends on how observable the system is.

4.5 The Simulator

```
1  Simulator ( Model, GoalSpec, InitialState, NumSolns ) {
2
3    // Constructor
4    Simulator. Model = Model
5    Simulator. GoalSpec = GoalSpec
6    Simulator. NumSolns = NumSolns
7    Simulator. CurrentTrajectories = initial trajectory from InitialState
8  }
9
10 Step ( Commands ) {
11 returns Observations
12
13   let NextTrajectories = FindModifiedKMostLikelyTrajectories
14     ( Simulator. Model, Simulator. GoalSpec, Simulator. CurrentTrajectories,
15       Commands, Simulator. NumSolns )
16   foreach NextT in NextTrajectories
17     let Observation = compute assignment to observables entailed by NextT's
18       current state and Model
19     insert Observation into Observations
20   endfor
21   Simulator. CurrentTrajectories = NextTrajectories
22   return Observations
23 }
```

Figure 4-6: The Simulator Pseudocode.

We now examine the Simulator in greater detail. The pseudocode given in Figure 4-6 shows a **Simulator()** constructor by which we initialize the Simulator and a **Step()** function. **Step()** is called on each iteration of the verifier with a list of commands and returns a list of observations. The Simulator maintains a set of the current most likely plant trajectories, which it updates at each time step. Its first action is to invoke our modified k Most Likely Trajectories algorithm (Line 6), described later on. It provides the algorithm with the current set of plant trajectories and commands, as well as the model and goal specification of the model-based pro-

gram, and obtains the set of next most likely program failure trajectories. For each next trajectory, it computes an assignment to the observable variables of the model that is entailed by the last state of the trajectory and the model (Line 17). It inserts this observation into a list of observations (Line 19). Finally, it updates the current trajectories (Line 21) and returns the list of observations to the verifier (Line 22). Figure 4-7 graphically illustrates one step of the Simulator. The Simulator receives a list of commands 1 through k, which are passed to the k current trajectories. Based on the new information, it generates the k next trajectories and their corresponding observations.

In our algorithm, we assume that the plant model is determinate, and therefore, a state uniquely determines an observation. However, if the plant model is an indeterminate, partial specification, then multiple observations may be consistent with a state, and the observations may have different likelihoods. Therefore, branching on unassigned observable variables with different probabilities could be a future extension to this algorithm.

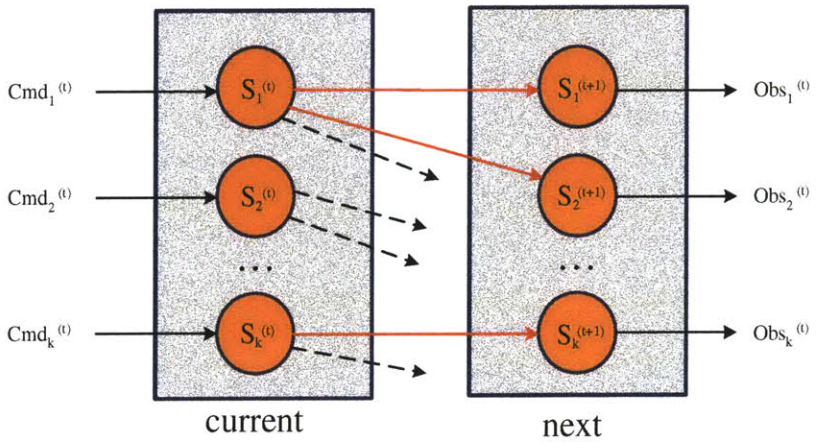


Figure 4-7: One step of the Simulator.

In Chapter 3 we motivated and described the k Most Likely Trajectories algorithm employed by Titan's Mode Estimation. We reuse k Best Trajectories ME in our simulator. Figure 4-8 gives the pseudocode incorporating our changes to the algorithm. First of all, we pass in a list of commands rather than a single command.

```
1   ModifiedFindKMostLikelyTrajectories ( Model, GoalSpec, CurrentTrajectories,
2     Commands, NumSolns )
3   returns NextTrajectories
4
5   let NextTrajectories = {}
6   let PriorityQueue = {}
7   foreach CurrentT in CurrentTrajectories
8       let Command = Commands[ CurrentT ]
9       compute the most likely transition from CurrentT's current state, enabled
10        by Command and Model, that doesn't satisfy GoalSpec
11      let NextT = CurrentT + target state of enabled transition
12      insert NextT into PriorityQueue
13  endfor
14  while PriorityQueue is non-empty
15      let T = pop most likely trajectory from PriorityQueue
16      insert T into NextTrajectories
17      if ( size of NextTrajectories == NumSolns )
18        return NextTrajectories
19      endif
20      let OrigCurrentT = T - last state of trajectory T
21      let Command = Commands[ OrigCurrentT ]
22      compute the next most likely transition from OrigCurrentT's current state,
23        enabled by Command and Model, that doesn't satisfy GoalSpec
24      let NextT = OrigCurrentT + target state of enabled transition
25      insert NextT into PriorityQueue
26  endwhile
27  return NextTrajectories
```

Figure 4-8: The Modified k Most Likely Trajectories Algorithm.

Since we are using Titan to individually propagate a set of trajectories rather than just one, we receive a corresponding number of commands. Second, we no longer pass in observations, since our objective, as a simulator, is to generate the observations from the commands. Finally, we consider the goal specification of the model-based program when determining which transitions are tracked. Since we are interested only in counterexamples, that is, trajectories that do not satisfy the goal specification, we disallow transitions to states that fulfill the goal of the program. The goal specification is a propositional state logic sentence; for example, a specification could be **(RoverTargetPosition = Reached)**. We take the negation of this sentence and add it to the logic constraints for the model. We determine enabled transitions based

on whether they satisfy the negation of this goal constraint, in addition to the constraints asserted by the modes and transitions of the model. Therefore the algorithm only generates the most likely trajectories that fail to achieve the program goals. As before, the cost of a trajectory is its computed probability and therefore, the probability of the execution it represents. The algorithm performs a greedy forward-directed search over the space of possible trajectories. At each iteration, best-first search is used to select the next set of trajectories.

Chapter 5

Verification of the Mars Entry Scenario

We have defined the verification problem we wish to solve and have introduced our algorithm. This chapter demonstrates verification on the Mars Entry model-based program, an entry scenario for a Mar lander spacecraft. Section 5.1 introduces the Mars Entry scenario. Sections 5.2 and 5.3 describe the inputs and outputs to the problem. Section 5.4 walks through the verification of the model-based program. The model-based program and its corresponding figures are taken from [11], with permission of the author Michel. D. Ingham.

5.1 The Model-based Program for the Mars Entry Scenario

Spacecraft are a natural application for model-based systems, as they must operate for long periods of time in unknown environments, with little human intervention. For example, one may concisely express the entry sequence for a Mars lander spacecraft using a model-based program, as described in [11]. In a typical scenario, the spacecraft begins a sequence of actions in preparation for its entry into the Martian atmosphere, as it nears the end of the cruise phase of its mission. As shown

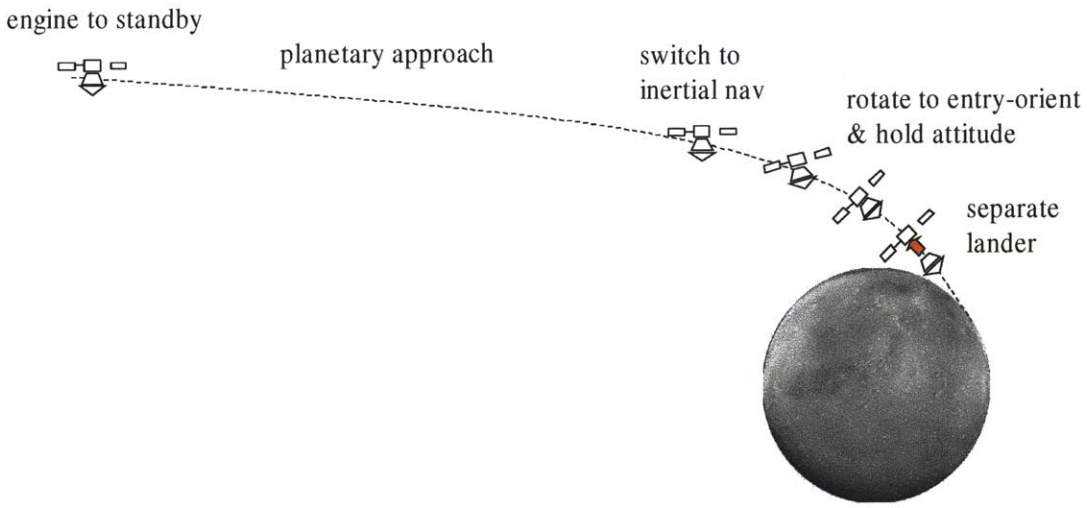


Figure 5-1: The Entry Sequence for a Mars Lander Spacecraft.

in Figure 5-1, it first turns its descent engine to standby mode, then switches from Earth-relative navigation to inertial navigation. It next rotates to and maintains its entry orientation. Once it has reached the proper orientation, the lander stage of the spacecraft separates from the cruise stage and descends towards its destination. Atmospheric entry is determined by a change in the spacecraft's acceleration, at which point the entry sequence ends. Each of the actions described can be modeled and controlled as hidden states, in accordance with the model-based programming paradigm.

5.2 Inputs

We now take a more detailed look at the inputs to the RMPLVerifier, which consist of a control program, a goal specification, a plant model, and a number of parameters that specify the search.

5.2.1 The Control Program

The control program directs the behavior of the system by interacting with the hidden state of the plant model. Figure 5-2 shows an example RMPL control program for the Mars lander entry sequence [11]. This program manipulates the hidden state of Engine, Nav, Att, Lander and Entry model components, which respectively represent the descent engine of the spacecraft, its navigation capability, its attitude, the lander stage, and the entry itself. First, the engine is turned to **Standby** mode (Line 2). Next, the craft changes to inertial navigation (Line 3). Then, while entry is not yet initiated, the attitude is set to always be **Entry-Orient** (Line 5) and the lander stage is detached (Line 6). The **EntrySequence()** procedure uses RMPL constructs to compactly specify the actions of the spacecraft during its entry phase.

As noted in Chapter 3, the intermediate representation of a control program is a Hierarchical Constraint Automaton (HCA) [20]. This compiled form is input to the verifier.

```
1 EntrySequence() {
2   Engine = Standby;
3   Nav = Inertial;
4    do {
5         always (Att = Entry-Orient),
6         when (Att = Entry-Orient)  donext (Lander = Separated)
7   }  watching (Entry = Initiated)
8 }
```

Figure 5-2: The RMPL Control Program for the Mars Entry Scenario.

5.2.2 The Goal Specification

As part of the verification process, we make explicit the mission of the program by asking the user to include a goal specification in the control program. A control program procedure is given a post-condition that indicates what should be true on the successful completion of that procedure. This specification is then automatically translated by the verifier into propositional state logic constraints and added to the

constraints expressing the plant model. Propositional state logic is the constraint system supported by RMPL. Consistent with our goal of easing the development process, we decided to express this knowledge as an addition to the control program. By adding RMPL post-conditions to each procedure, we add both functionality and documentation.

Intuitively, the definition of success for the Mars Entry control program is the separation of the lander, since that indicates the conclusion of the entry sequence. Therefore its goal specification could be expressed as the propositional state logic sentence **Lander = Separated**. Figure 5-3 shows the control program enhanced by a goal specification.

```
1  EntrySequence() {
2     postcondition (Lander = Separated);
3
4     Engine = Standby;
5     Nav = Inertial;
6      do {
7           always (Att = Entry-Orient),
8           when (Att = Entry-Orient)  donext (Lander = Separated)
9     }  watching (Entry = Initiated)
10 }
```

Figure 5-3: The RMPL Control Program and Goal Specification for the Mars Entry Scenario.

5.2.3 The Plant Model

The plant model, the second component of the model-based program, represents the possible states of the system hardware. Continuing with our previous example, we may model the Mars lander spacecraft with a number of components that can then be controlled by the control program [11]. Figure 5-4 shows all the components of the model, along with their interconnections. As we can see, the engine is connected to two fuel tanks via valves. The PDE can command the engine and valves. We give a brief description of each component. The Att component (Figure 5-5) is a qualitative

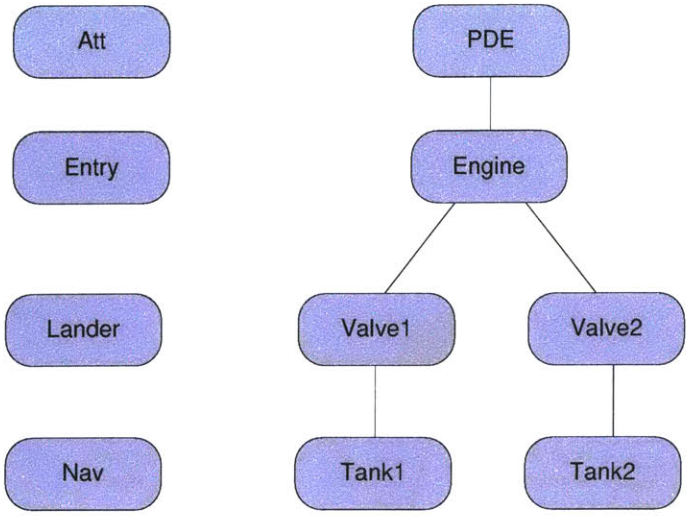


Figure 5-4: The Plant Model for the Mars Entry Scenario.

model of the spacecraft's attitude. The Engine component (Figure 5-6) models the spacecraft's engine. The Entry component (Figure 5-7) gives the entry status. The Lander component (Figure 5-8) represents a simple lander/cruise stage separation pyro mechanism. The Nav component (Figure 5-9) models the craft's navigation. The PDE component (Figure 5-10) represents the propulsion drive electronics. The Tank component (Figure 5-11) models a propellant tank. The Valve component (Figure 5-12) models a simple valve. The compiled Concurrent Constraint Automaton (CCA) representation of the plant model is input to the verifier.

5.2.4 Other Parameters

In addition to the model-based program, enhanced by the goal specification, the verifier takes in a number of parameters that specify the search:

- The initial state of the plant model. This consists of assignments to the component mode variables.
- The name of the control program procedure to execute.
- How many time steps of the program's execution the verifier should examine.

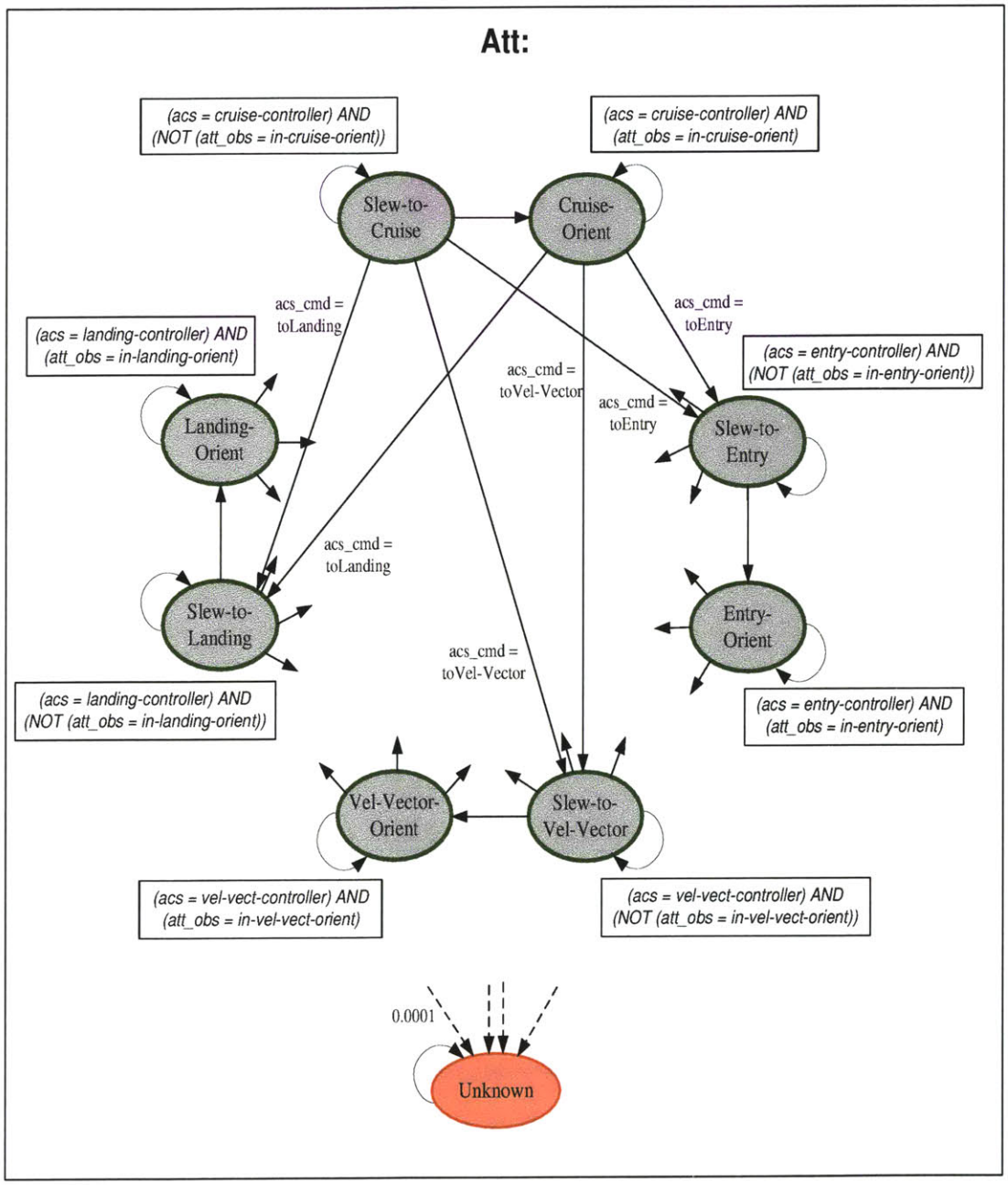


Figure 5-5: The Att Component.

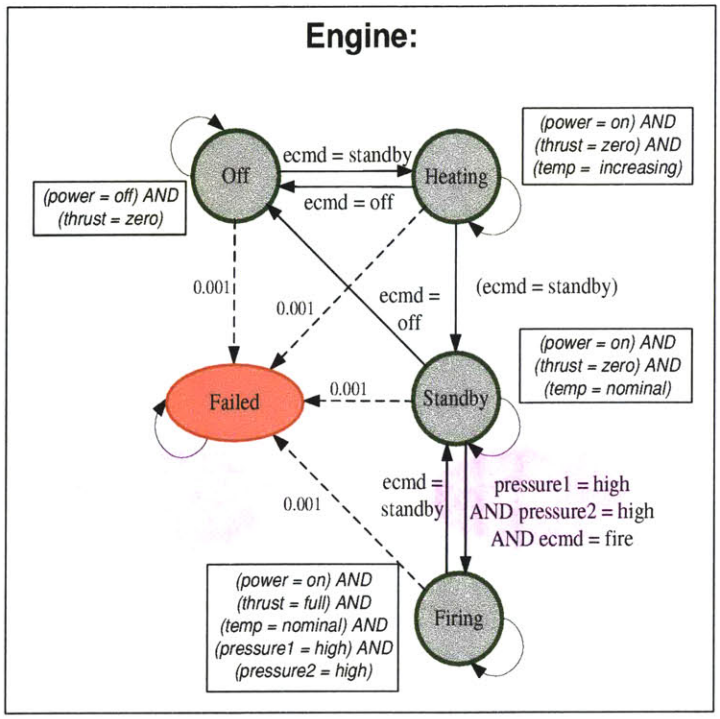


Figure 5-6: The Engine Component.

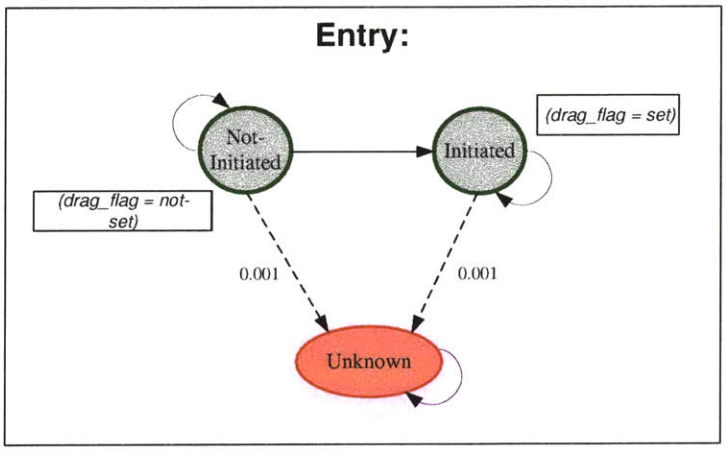


Figure 5-7: The Entry Component.

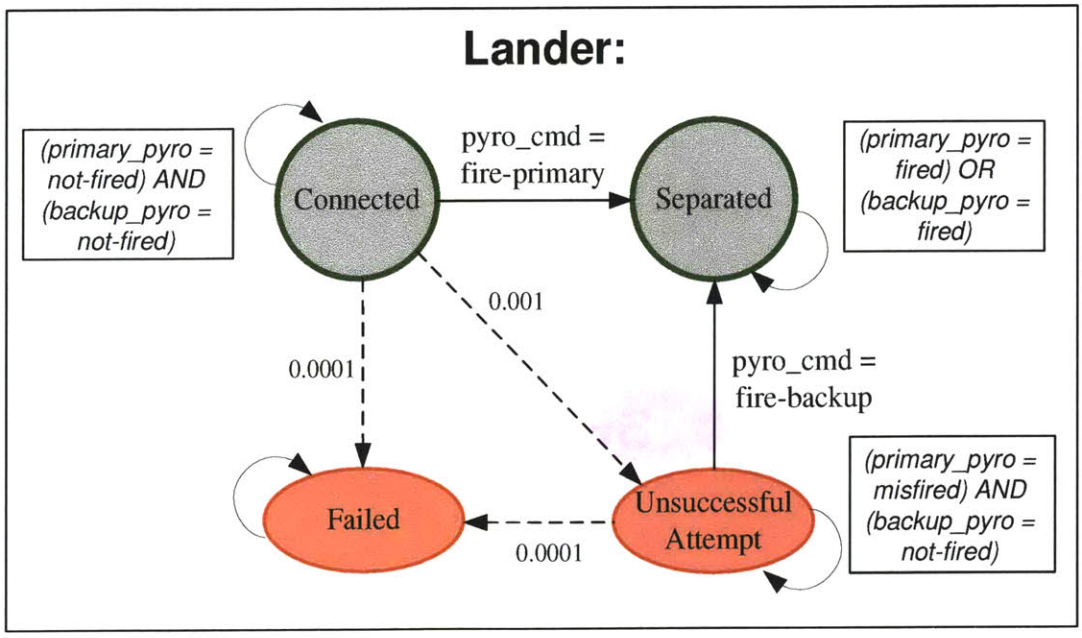


Figure 5-8: The Lander Component.

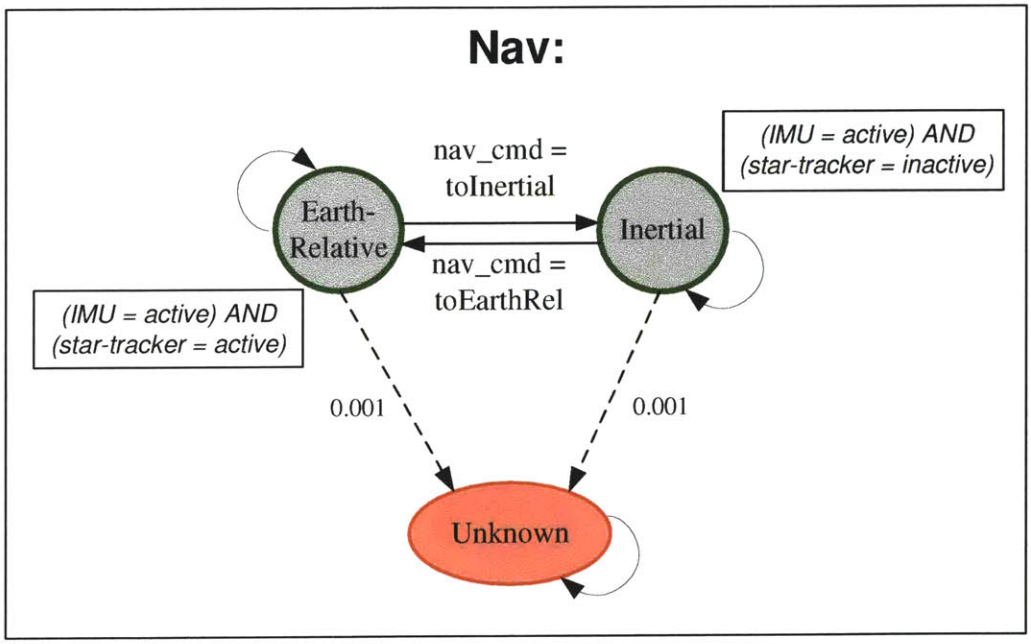


Figure 5-9: The Nav Component.

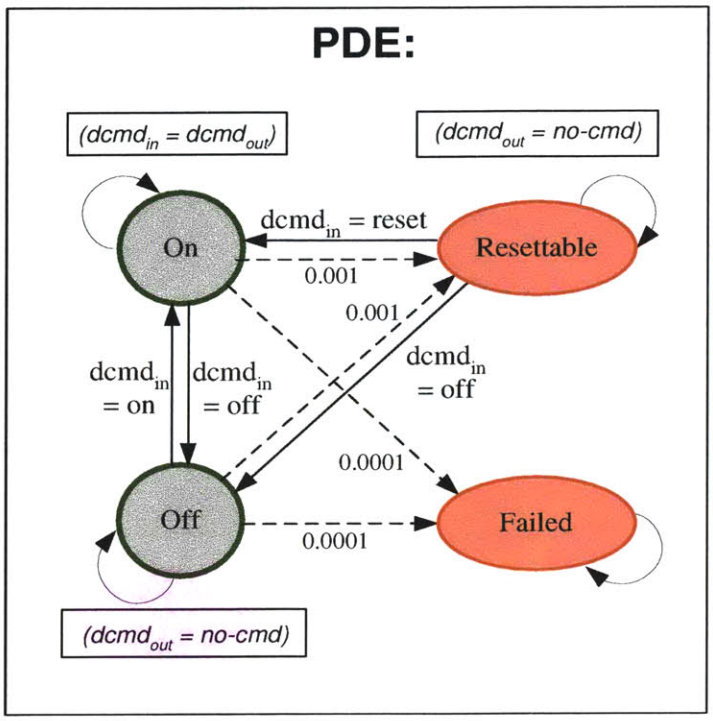


Figure 5-10: The PDE Component.

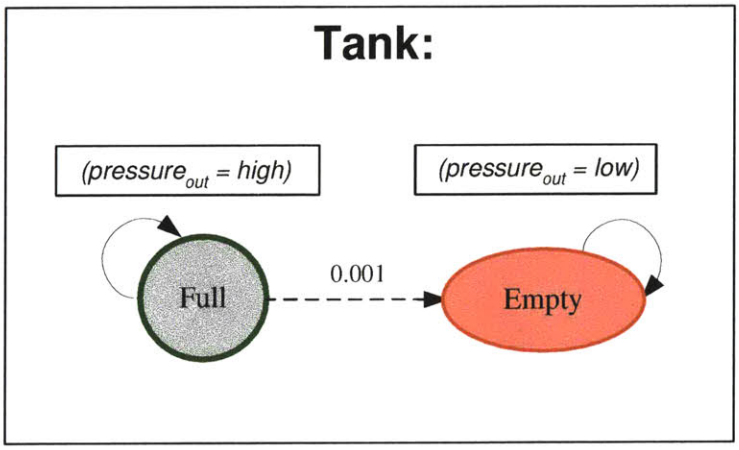


Figure 5-11: The Tank Component.

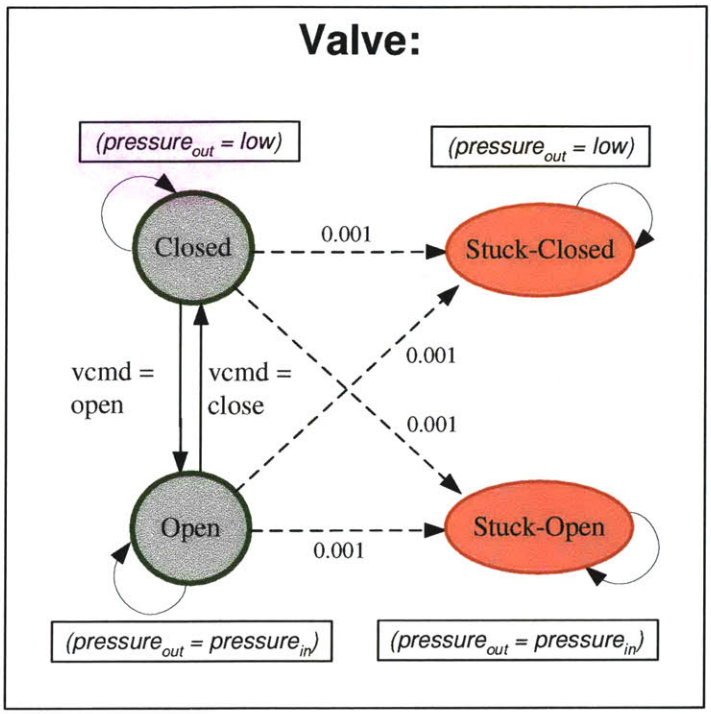


Figure 5-12: The Valve Component.

- How many program failure trajectories the verifier should find.

5.3 Outputs

The verifier produces as output a list of counterexamples ranked by probability, from highest to lowest. A counterexample is a plant state trajectory that does not achieve the goal specification within the specified number of steps. Figure 5-13 shows the first counterexample from a run of the RMPLVerifier on the Mars Entry model-based program, with control program procedure **EntrySequence()** and **LANDER1.MODE = SEPARATED** as the definition of success. The initial state of the plant is given on Lines 9-18. The number of time steps examined is seven, and the number of program failure trajectories returned is five. The first counterexample, which is also the most likely, proceeds through the entry sequence of the spacecraft, taking nominal transitions. At the last step, it takes a transition to a failure state.

We look at this first counterexample in detail. The probability of the most likely program failure trajectory is given as 5.41851e-05 (Line 5). At Time Step 0, we see the initial plant state, which is provided as input to the RMPLVerifier. Next, we begin execution of the model-based program at Time Step 1. The output for a time step includes the observations received during that step (Lines 22-35), the command issued by Titan's Mode Reconfiguration (Line 37-38), and the plant state at the end of the step (40-42). Each of these is an assignment to the appropriate variables of the plant model. In this particular counterexample, PDE, the propulsion drive electronics component of the spacecraft, begins in the off state (Line 15). MR issues a command to turn PDE on during the first time step. The new plant state shows that the command was successful (Line 42). The values of the other mode variables stay the same.

We briefly go through each remaining time step. At Time Step 2, the observations have not changed. The command is to turn the engine to standby. The next plant state shows that the engine is heating, which is an intermediate step to standby. At Time Step 3, the observations show that the engine temperature is increasing and

```
1  5 counterexamples were found:
2
3  # Counterexample 1 #
4
5  This trajectory has probability = 5.41851e-05
6
7  At the completion of Time Step 0, the plant state is:
8
9  ENGINE1.MODE = OFF
10 NAV1.MODE = EARTH-RELATIVE
11 TANK1.MODE = FULL
12 TANK2.MODE = FULL
13 VALVE1.MODE = CLOSED
14 VALVE2.MODE = CLOSED
15 PDE1.MODE = OFF
16 ATT1.MODE = CRUISE-ORIENT
17 LANDER1.MODE = CONNECTED
18 ENTRY1.MODE = NOT-INITIATED
19 ----------
20 Executing Time Step 1 ...
21
22 The model-based executive received the following changed observations:
23 NAV1.IMU-STATUS = ACTIVE
24 ATT1.ACS-OBS = IN-CRUISE
25 LANDER1.BACKUP-PYRO = NOT-FIRED
26 LANDER1.PRIMARY-PYRO = NOT-FIRED
27 NAV1.ST-STATUS = ACTIVE
28 ENTRY1.ACCEL-OBS = ZERO
29 TANK2.PRESSURE-OUT = HIGH
30 ATT1.ACS-CTRL = CRUISE
31 ENGINE1.POWER = OFF
32 ENGINE1.THRUST = ZERO
33 VALVE2.PRESSURE-OUT = LOW
34 VALVE1.PRESSURE-OUT = LOW
35 TANK1.PRESSURE-OUT = HIGH
36
37 The model-based executive issued the following command:
38 PDE1.PDE-CMD-IN = ON
39
40 At the completion of Time Step 1, the changes to the plant state are:
41
42 PDE1.MODE = ON
43 ----------
44 Executing Time Step 2 ...
45
46 The model-based executive received the following changed observations:
47
48
49 The model-based executive issued the following command:
50 PDE1.ENGINE-CMD-IN = STANDBY
51
52 At the completion of Time Step 2, the changes to the plant state are:
53
54 ENGINE1.MODE = HEATING
```

Figure 5-13: The Verification Output for the Mars Entry Scenario.

```
1  ------------------------------------------------------
2  Executing Time Step 3 ...
3
4  The model-based executive received the following changed observations:
5  ENGINE1.TEMP = INCREASING
6  ENGINE1.POWER = ON
7
8  The model-based executive issued the following command:
9  PDE1.ENGINE-CMD-IN = STANDBY
10
11 At the completion of Time Step 3, the changes to the plant state are:
12
13 ENGINE1.MODE = STANDBY
14 ------------------------------------------------------
15 Executing Time Step 4 ...
16
17 The model-based executive received the following changed observations:
18 ENGINE1.TEMP = NOMINAL
19
20 The model-based executive issued the following command:
21 NAV1.NAV-CMD = TOINERTIAL
22
23 At the completion of Time Step 4, the changes to the plant state are:
24
25 NAV1.MODE = INERTIAL
26 ------------------------------------------------------
27 Executing Time Step 5 ...
28
29 The model-based executive received the following changed observations:
30 NAV1.ST-STATUS = INACTIVE
31 ENGINE1.POWER = ON
32
33 The model-based executive issued the following command:
34 ATT1.ACS-CMD = TO-ENTRY
35
36 At the completion of Time Step 5, the changes to the plant state are:
37
38 ATT1.MODE = SLEW-TO-ENTRY-ORIENT
39 ------------------------------------------------------
40 Executing Time Step 6 ...
41
42 The model-based executive received the following changed observations:
43 ATT1.ACS-CTRL = ENTRY
44
45 The model-based executive issued the following command:
46 No new command was necessary.
47
48 At the completion of Time Step 6, the changes to the plant state are:
49
50 ATT1.MODE = ENTRY-ORIENT
51 ------------------------------------------------------
52 Executing Time Step 7 ...
53
54 The model-based executive received the following changed observations:
55 ATT1.ACS-OBS = IN-ENTRY
56
57 The model-based executive issued the following command:
58 LANDER1.LANDER-PYRO-CMD = FIRE-PRIMARY
59
60 At the completion of Time Step 7, the changes to the plant state are:
61
62 LANDER1.MODE = UNSUCCESSFUL-ATTEMPT
63 ------------------------------------------------------
```

Figure 5-14: The Verification Output for the Mars Entry Scenario (Continued).

the power is on and the engine command is still standby. The plant state now shows that the engine has achieved standby mode. At Time Step 4, the observations are that the engine temperature is nominal. Consistent with the control program, the next command is to switch navigation to inertial mode. The plant state shows that this was successful. At Time Step 5, the navigation ST instrument status is inactive and the engine power is on and the attitude command was to set the orientation to **ToEntry**. The plant state showed that the spacecraft was slewing to the entry orientation. At Time Step 6, the attitude control was observed to be **Entry**. There was no command issued. The plant state showed that the attitude had been changed to the entry orientation. Finally, at Time Step 7, the attitude was observed to be in the in-entry orientation, and that the command was to fire the primary lander pyro. The plant state showed that the lander had unsuccessfully tried to separate.

We have examined the first plant trajectory output by the verifier in the Mars Entry scenario. In the next section, we see how the set of trajectories is generated.

5.4 Walkthrough of the Algorithm

We run the **EntrySequence()** control program and its corresponding spacecraft model on the RMPLVerifier. The horizon, which determines the number of time steps examined, is set to seven, and the number of state trajectories tracked and returned is set to five. Figures 5-15 and 5-16 show the state trajectories tracked by the verifier at each time step. Figure 5-15 covers time steps 0 to 3, and Figure 5-16 covers 4 to 7. Each rounded rectangle in the figure represents a plant state. Lines connecting the states represent plant state trajectories. Each level in the tree represents a time step. The states on a level are ordered from left to right, with the left-most state being the most likely at that point in time. We highlight any state assignment in a trajectory that has changed in the last time step. The figure also shows commands issued at each step.

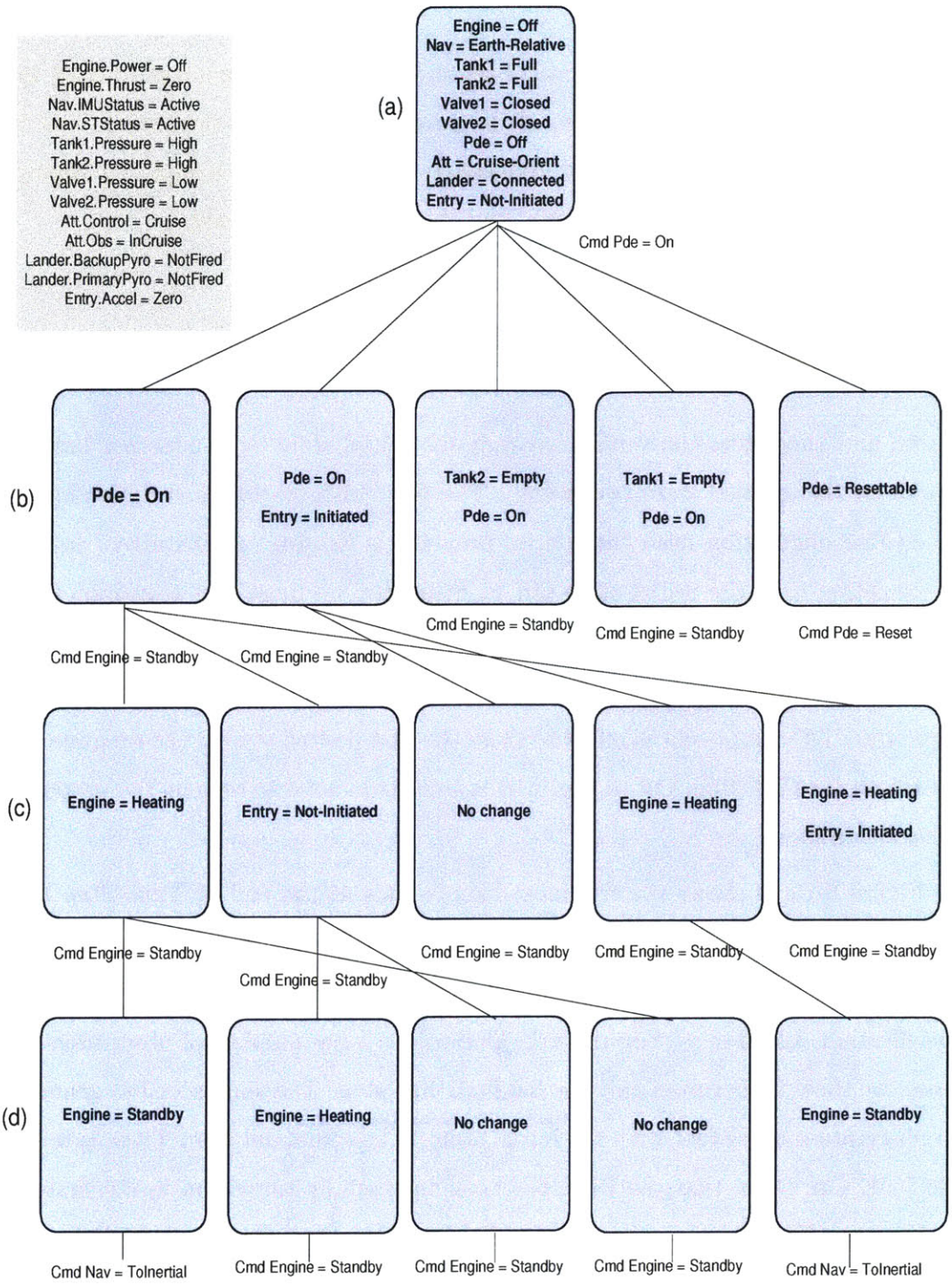


Figure 5-15: Verification of the Mars Entry Scenario. (a) The Initial State. (b)-(d) Time Steps 1-3.

5.4.1 Time Step 0

Figure 5-15(a) shows the initial state of the program as an assignment to state variables. In the beginning, the spacecraft is traveling towards its destination and has yet to begin entry into the Martian atmosphere. Therefore, for example, its engine is off, navigation is still relative to earth, and the lander is connected.

5.4.2 Time Step 1

In the first time step, the verifier expands the initial state by conducting best-first search on the enabled state transitions. The first configuration goal determined by Mode Reconfiguration from the control program is **Engine = Standby**. In order to transition from the initial state **Off** to **Standby**, we must first transition to the intermediate **Heating** mode of the engine. This may be accomplished by giving the engine the **Standby** command. However, we cannot issue any engine commands while the PDE component is off. Therefore the first desired step of the program is to transition the PDE from **Off** to **On**. MR issues a command to turn on the propulsion drive electronics.

Figure 5-15(b) shows the five most likely states at the end of Time Step 1. In practice, many more states are reachable, but the verifier prunes these during its best-first search. The verifier also will prune any trajectory that satisfies the goal specification, **Lander = Separated**. Due to the large number of observable variables, we show their values only for the first time step. The simulator has generated an observation consistent with the initial state. The command from Titan is to turn the PDE **On**. Note that the PDE has been successfully turned on in the first four trajectories. The fifth trajectory goes to **Resettable**, a failure mode of PDE. In three child trajectories, other state values have changed as well. The second trajectory also decides that entry has been initiated. The third and fourth trajectories think a tank is empty. These are less likely scenarios than the first trajectory. At the end of the first time step, we thus have four trajectories that turn the PDE on.

5.4.3 Time Step 2

Now that the PDE is on, we are ready to issue commands to the engine. We are still pursuing the control program goal to set **Engine = Standby**. During Time Step 2 (Figure 5-15(c)), we issue a command to the first four trajectories to turn on the engine. We issue a command to reset the PDE to the fifth trajectory, as it has not yet turned the PDE on. We see that some trajectories succeed in setting **Engine** to **Heating**, while others do not. An important fact to note about Time Step 2 is that the most likely trajectories are derived from only the first two previous trajectories; the remainder are pruned.

5.4.4 The Remaining Time Steps

We proceed in a similar manner through each time step (Figures 5-15(d) and 5-16(a)-(d)), issuing configuration goals to switch to inertial navigation and to set the attitude to the entry orientation of the spacecraft and successfully and unsuccessfully completing commands. Eventually we reach Time Step 7 (Figure 5-16(d)), where one trajectory receives a command to fire the primary lander pyro. At this stage, our goal specification for the program, **Lander = Separated**, comes into play, preventing the lander from separating. The state for the first trajectory reflects the fact that the attempt was unsuccessful. Beginning at Time Step 5, we also get trajectories that failed to achieve their configuration goals. Mode reconfiguration issues an error when a trajectory has failed to achieve a configuration goal within a number of steps.

At the conclusion of the verification algorithm, we are left with the five most likely program failure trajectories in the bound of seven time steps. The trajectories are labeled with their probabilities in Figure 5-16(d). The most likely trajectory proceeds nominally through the Mars entry sequence until it reaches the lander separation state, when it fails to separate. We would expect the most likely state to be the separation of the lander. However, that is not allowed by our goal specification, so the lander being unsuccessful is the next most likely state for that trajectory. The third trajectory is similar, only it takes longer to set the attitude to the entry

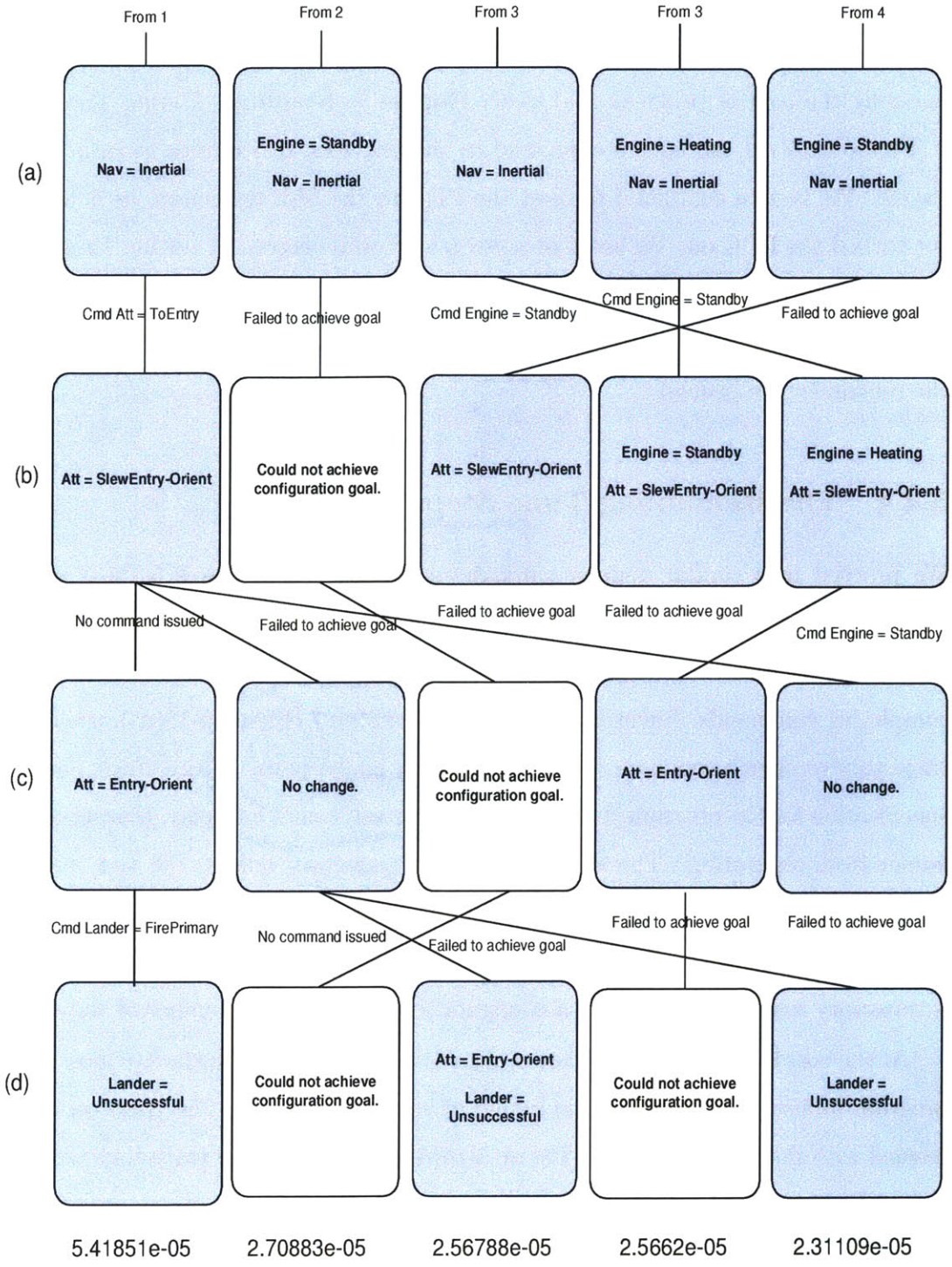


Figure 5-16: Verification of the Mars Entry Scenario (cont'd). (a)-(d) Time Steps 4-7.

orientation. The fifth trajectory has not yet achieved the entry orientation when the verification terminates. For the last two time steps, this trajectory receives no command from MR. Instead, Mode Reconfiguration states that is waiting for an appropriate observation. It does not receive the necessary observation to achieve the entry orientation goal by Time Step 7. The second and fourth trajectories fail to achieve their configuration goals. The second trajectory first gets an error message from MR at Time Step 4. For the last three time steps before, the command is to turn the engine to standby. However, this state is not achieved by the plant. Therefore, Mode Reconfiguration gives an error saying that it cannot achieve this configuration goal. The fourth trajectory fails for the same reason, only it first gets an error from MR in Time Step 6. The results described above are illustrative of the different categories of failure trajectories the verification process may reveal. The results are particularly useful when considering complex models like the Mars Entry scenario, where the possible states, commands and observations are far too many to enumerate by hand.

Chapter 6

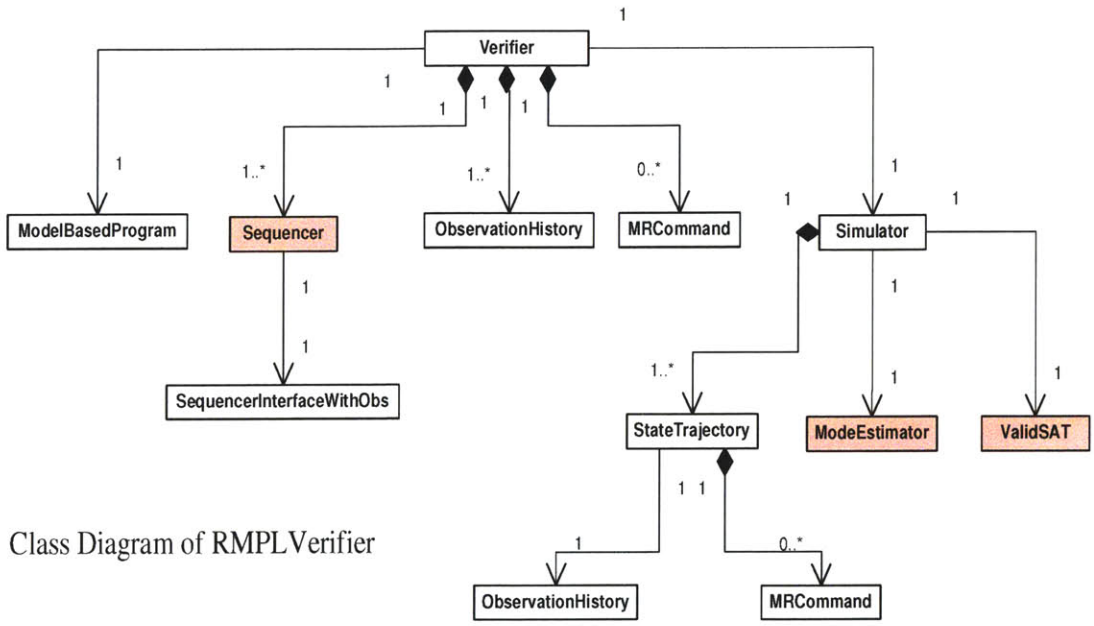
Implementation

RMPLVerifier is implemented as a C++ program that uses the C++ API of Titan 1.2, the current version of Titan. In this chapter we describe the implementation. We begin with a general overview of the system architecture in Section 6.1. Section 6.2 describes components of Titan used by RMPLVerifier. Section 6.3 takes a detailed look at key components of RMPLVerifier and outlines the main implementation issues. In particular, we examine the Verifier, Simulator, and SequencerInterfaceWithObs classes, which encapsulate the core functionality of the software.

6.1 System Architecture

Figure 6-1 shows the architecture of the system as a Unified Modeling Language (UML) class diagram. Each block in the diagram represents a class. A relationship between classes is described with a connecting link. An arrow on the link shows the direction of the association. For example, the arrow from Verifier to ModelBasedProgram indicates that Verifier can be queried about its ModelBasedProgram, but not vice versa. A diamond end to the link indicates that the class contains a collection of the class on the other end of the link. For example, a StateTrajectory contains a collection of MRCommands. The end of the link shows the multiplicity of the class, which is the number of possible instances of the class that can be associated with the class on the other end. The multiplicity can be:

- **0...1** - Zero or one instance.
- **0...*** - Any number of instances, including zero.
- **1** - One instance.
- **1...*** - One or more instances.



Class Diagram of RMPLVerifier

Figure 6-1: The UML class diagram for RMPLVerifier.

The shaded blocks indicate classes of Titan used by RMPLVerifier.

Verifier is the root of the class hierarchy. It takes a ModelBasedProgram and the search parameters as input and produces a ranked list of trajectories as output. ModelBasedProgram represents the model-based program and contains objects representing the control program and plant model, the control program procedure, and the initial state of the model. Verifier handles the interaction between Titan and Simulator. It maintains a list of the current MRCommands to be provided from Titan to Simulator. It maintains another list of the current ObservationHistory objects to be provided from Simulator to Titan. Verifier interfaces with the Titan system using the Sequencer class. Sequencer is the top-level object for Titan, consisting of

the control sequencer and deductive controller. Verifier constructs Sequencers using SequencerInterfaceWithObs, an interface that allows one to receive commands from Titan and inject observations from Simulator during execution of the model-based program.

Simulator implements a simulator that maintains a list of the current StateTrajectory's, ranked by probability. StateTrajectory represents a plant state trajectory and contains the ObservationHistory of the trajectory, as well as the command history as a list of MRCommands. ObservationHistory is a list of the observations at each time step. MRCommand represents a command received from Titan's Mode Reconfiguration. Simulator queries the ModeEstimator class to generate the next set of StateTrajectory's. ModeEstimator is the Mode Estimation component of Titan. Simulator then uses ValidSAT [5] to find observations consistent with the next states. ValidSAT is a satisfiability engine available as part of the Titan API.

6.2 Relevant Components of Titan

RMPLVerifier calls on the Titan executive to execute model-based programs. Here we describe key components of Titan used in the implementation.

6.2.1 Sequencer

Sequencer implements Titan's control sequencer and deductive controller and is used to execute a model-based program. It contains all the subsystems of Titan, including Mode Estimation and Mode Reconfiguration; the ModeEstimator object is its subcomponent. It also contains the Titan data structures representing the model-based program and is initialized with the initial state and a control program procedure. Sequencer maintains all the state information during execution, including the trajectories for the plant model and control program. Starting the Sequencer begins the execution of the model-based program by producing the initial state estimate. Stepping the Sequencer executes the program for one time step, during which the next configuration goal is generated from the control program, Mode Reconfiguration

issues the next command, and Mode Estimation computes the next estimated states.

6.2.2 ModeEstimator

ModeEstimator implements Titan's Mode Estimation subsystem. It produces a single most likely state estimate that is used by the rest of Titan. The current version of ModeEstimator in Titan 1.2 implements the k Most Likely Trajectories algorithm. We use ModeEstimator in two capacities. We use it indirectly for execution as part of the overall Titan system. In this case, ModeEstimator applies the same command to all state trajectories. We also use a separate instance for simulation. In this case, it applies an individual command to each state trajectory.

6.2.3 ValidSAT

ValidSAT [5] is a satisfiability engine. It is able to classify theories as either consistent, inconsistent, or valid. The engine uses a clause-directed approach along with finite-domain variables to efficiently test for validity. These techniques allow it to test for validity without assigning a value to every variable. ValidSAT is also able to extract the minimal valid and minimal inconsistent assignment it found while classifying the theory. We use ValidSAT in the simulator to return the first minimal valid solution to a satisfiability problem where the observable variables are the decision variables and the constraints are the constraints on the modes of the plant model, along with the mode values of the current state. This solution is the consistent observation for that state.

6.3 Key Components of RMPLVerifier

We now describe the Verifier, Simulator, and SequencerInterfaceWithObs classes in more detail.

6.3.1 Verifier

Verifier is called by VerifierCLI, the top-level program of RMPLVerifier. The user invokes VerifierCLI and enters the inputs to the verification problem via it's command-line interface. These inputs are:

- -h Specifies the name of the file containing the HCA of the enhanced control program.
- -c Specifies the name of the file containing the CCA of the plant model.
- -i Specifies the name of the file containing the initial state of the plant model.
- -p Specifies the name of the control program procedure to execute.
- -n Specifies how many time steps of the program's execution the verifier should examine.
- -k Specifies how many program failure trajectories the verifier should find.

VerifierCLI loads the inputs and creates the ModelBasedProgram object used in the rest of the program. It prints the state trajectories obtained from Verifier to an output log file, as a list of counterexamples ranked by probability.

Verifier takes the ModelBasedProgram and search parameters as input and generates an ordered list of plant state trajectories as output. It keeps a reference to an ObservationHistory for each current state trajectory. Initially, there are no state trajectories, so this list contains a single empty ObservationHistory. Verifier also keeps a list of the MRCommands received during the current time step, one for each state trajectory. Initially, this list is empty. Verifier performs iterations, each corresponding to a time step, until the time horizon has been reached. It takes the following actions during each iteration.

First, it gets the new list of commands from Titan. For each current trajectory, it needs an instance of Titan that contains all the state for that trajectory. That instance will then output a new command for the trajectory given the latest observation

from the simulator. Verifier loops over the list of ObservationHistorys, each of which contains the latest observation for its associated trajectory. For each ObservationHistory, it recreates the Sequencer object for the state trajectory it represents. It does so by creating a new Sequencer object based on the ModelBasedProgram and stepping it through the model-based program for the necessary number of steps. At each step, it uses SequencerInterfaceWithObs to input the appropriate observation from the observation history. The command history is automatically recreated. Once it has the Sequencer representing the current trajectory, it obtains the new command issued by Mode Reconfiguration based on the current state. Verifier stores the command in its list of commands. Next, it passes the current list of commands to Simulator and gets the new list of ObservationHistorys. Once the time horizon has been reached, Verifier returns the list of current StateTrajectory's from Simulator.

6.3.2 Simulator

Simulator is initialized with the plant model and goal specification of the model-based program. It maintains a list of the current StateTrajectory's. The following actions occur for each simulator step.

Simulator begins by calling ModeEstimator with the latest list of commands it has received. ModeEstimator returns the list of next states. Simulator gives ModeEstimator a modified version of the plant model as input. The plant model is essentially a set of propositional state logic constraints. Simulator creates a logic sentence that is the negation of the goal specification for the control program procedure. The specification, as procedure post-conditions, is loaded automatically as part of the control program at the beginning of the program. Simulator inserts it into the model. This ensures that ModeEstimator only returns estimated states that do not satisfy the specification.

Simulator then computes an observation consistent with each next state using ValidSAT. A state, consisting of the full assignment to the state variables, is provided to ValidSAT along with the modal constraints from the plant model. The observable variables are set as the decision variables in the satisfiability problem. ValidSAT

returns the first minimal valid assignment to observable variables that satisfies the constraints. Simulator updates each StateTrajectory with the next state estimate, the probability of the trajectory from ModeEstimator, and the new command and observation.

6.3.3 SequencerInterfaceWithObs

SequencerInterfaceWithObs is an interface that allows Verifier to receive commands from and give observations to Titan. SequencerInterfaceWithObs is passed as a constructor argument to the Sequencer. Ordinarily, Sequencer is created with SequencerInterface, which passes commands directly from Mode Reconfiguration to Mode Estimation and has no facility for injecting observations. By creating SequencerInterfaceWithObs, a subclass of SequencerInterface, we were able to obtain commands from Mode Reconfiguration and inject observations from an observation history. By constructing the class with an observation history, we are able to recreate the Sequencer state for a particular program state trajectory. SequencerInterfaceWithObs can be queried for the newest command issued by Mode Reconfiguration.

6.4 Implementation Issues

The main implementation challenge lay in maintaining the list of program state trajectories using Sequencer. A Sequencer instance holds all the state for Titan. Therefore it seemed a natural choice to store the information for a program state trajectory during the search. However, in practice, unexpected errors occur when multiple Sequencer objects exist and are running at the same time. Therefore we were unable to keep a list of Sequencer objects as representations of state trajectories. Another factor in this decision was the fact that a Sequencer has no copy constructor. This is reasonable for Titan, as there is ordinarily no reason to copy a Sequencer, and it would undoubtedly be an expensive operation. However, it also meant that we could not create two new trajectories from the same parent trajectory by simply copying the parent node's Sequencer and stepping it once. Our solution for both of these

issues was to store enough information in a StateTrajectory object that we were able to recreate the Sequencer for its corresponding program state trajectory when needed. In particular, we stored the observation history for each time step. Unfortunately, the Sequencer must step through the whole execution of the model-based program, duplicating work we have done before. Future optimizations could focus on storing some of the state of the Sequencer in order to reduce the time needed.

6.5 Summary

In this chapter we presented the implementation of the RMPLVerifier. In the next chapter we look at results and give ideas for future work.

Chapter 7

Results and Conclusions

This thesis presented a novel verification approach embodied in RMPLVerifier, a tool for verification of RMPL model-based programs. Our approach provides three key capabilities. First, we provide the ability to verify a stochastic system that encodes both off-nominal and nominal scenarios. Second, we provide a capability for verifying executable specifications that are fault-aware. Third, we verify these specifications through execution.

This chapter presents the performance of RMPLVerifier on the Mars Entry model-based program (Section 7.1). We conclude by discussing future work (Section 7.2).

7.1 Performance

We measured the performance of the RMPLVerifier with respect to time. The program was tested on an Intel(R) Xeon(TM) 1.7 GHz processor with 500 MB of RAM running the Debian Linux 2.4.23 operating system. It was run on the Mars Entry model-based program with different combinations of search parameters. The Mars Entry model is composed of 8 component models. It has 7 control variables, 15 observable variables, 10 state variables, and 10 dependent variables, for a total of 42 variables. It has 179 transitions.

For our first benchmark, we varied the number of time steps the verifier examined while keeping the number of tracked trajectories constant. Table 7.1 shows the time

Number of Time Steps	Number of Trajectories	Average Time (seconds)
1	5	0.396
2	5	2.898
3	5	5.96
4	5	10.066
5	5	15.758
6	5	22.462
7	5	29.511

Table 7.1: Performance of RMPLVerifier on the Mars Entry model-based program with respect to the number of time steps.

Number of Time Steps	Number of Trajectories	Average Time (seconds)
1	1	0.349
1	2	0.36
1	3	0.363
1	4	0.388
1	5	0.412
1	6	0.426
1	7	0.441

Table 7.2: Performance of RMPLVerifier on the Mars Entry model-based program with respect to the number of trajectories.

performance of the program in this case. For each test, we measured the total number of seconds that the process used directly in user mode and the total number of seconds used by the system on its behalf in kernel mode. We added these two numbers to obtain the total time the process ran. We averaged the total time over ten runs to compute the average time for the test case.

For our second benchmark, we varied the number of trajectories to find while keeping the number of time steps constant. Table 7.2 shows the time performance of the program. We computed the average time in the same manner.

The data we collected agreed with our intuition on the performance of the program. As we increase the number of trajectories, the time for the program to complete increases in a roughly linear fashion. This corresponds to the behavior we would expect for Mode Estimation, which is the computational core of the simulator. However, as we increase the number of time steps, the time for the program to complete increases exponentially. This is an artifact of the implementation. We can understand

why this happens by looking at the process by which the verifier computes the next set of Titan trajectories from the current one. Let us define the time needed to extend each of the current set of Titan trajectories by one Titan step as a constant T. Therefore, the time taken to complete one time step is T added to the time to recreate the current Titan trajectories. This can be described by the following recurrence:

$$Step_n = T + Step_{n-1} + Step_{n-2} + ... + Step_1 = T * 2^{n-1}$$

Therefore, the time to execute a program that examines n steps is:

$$Time_n = \sum_{i=1}^{n} Step_i = T * (2^n - 1)$$

This performance bottleneck is strictly an implementation issue, which can be addressed as a software engineering task. We discuss ways to improve it in the next section.

7.2 Future Work

We discuss areas of future research in the treatment of observations, performance, and the presentation of results.

7.2.1 Observations

The treatment of observations in the verifier could be improved. Currently the simulator returns the first observation consistent with a next state. However, in reality, there may be a number of equally valid observations. A future version of the verifier could reflect this by branching on all consistent observations. In addition, we have assumed that all observations have equal likelihood, which is not the case. A better strategy would account for the observation probability in the cost of a trajectory. Note that observation probabilities are not currently part of RMPL specifications.

7.2.2 Performance

We have adopted the strategy of local beam search for our algorithm. At each step, we generate the k best successors of the current k plant states. Initially, we considered using simple best-first search for our algorithm. However, we realized that generating all the possible next states for a current plant state would quickly grow intractable in terms of the size of the queue we would have to maintain. The queue size remains constant in our current search strategy. However, a disadvantage of our approach is that we have no way to steer the search towards non-achievement of the goal, since we only use local information to make decisions. Therefore, a future version of the algorithm could employ a different search strategy, one that uses an admissible heuristic to calculate the cost to reach the negation of the goal.

The performance of our implementation could be improved by finding an alternate strategy for saving and restoring the state of Titan for a trajectory. Currently it is not possible to save this state as a Sequencer object because of buggy behavior that occurs when multiple instances of the Sequencer are run soon after each other. One strategy would be to work with Titan's developers to eliminate this problem. However, since RMPLVerifier uses Titan as a client, it might be more practical to find a solution that uses Titan as is. An alternate strategy would be to save our own representation of Titan's state at any point in time and use it to recreate a Titan instance.

7.2.3 Presentation of Results

There are two main ways that the presentation of results to the user could be improved. The first is the addition of a graphical user interface in addition to the command-line interface that is currently available. Desired features of this interface could include the ability to visualize, step through, and play back trajectories. Another is the abstraction of the trajectories returned. It is possible that some trajectories returned will have many states in common. Trajectories could be grouped together by similarity and common points of failure to increase the relevance of the results to the user.

7.3 Summary

The verification and validation of model-based systems is an area that has been little explored. However, V & V is key to proving the viability of model-based autonomy as an alternative to human control on missions. This thesis brings us a step closer to achieving that goal. We have presented a verification approach for model-based programs. Our approach provides three capabilities. We give the ability to verify stochastic systems that define both nominal and failure scenarios. We enable verification of executable specifications that are fault-aware. Finally, we verify these specifications through execution.

Bibliography

[1] Gerard Berry and Georges Gonthier. The Esterel Synchronous Programming Language: Design, Semantics, Implementation. *Science of Computer Programming*, 19(2):87–152, Jan 1992.

[2] G. Brat, D. Giannakopoulou, A. Goldberg, K. Havelund, M. Lowry, C. Pasareanu, A. Venet, W. Visser, and R. Washington. Experimental Evaluation of Verification and Validation Tools on Martian Rover Software. In *CMU/SEI Software Model Checking Workshop*, Pittsburg, USA, March 2003.

[3] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. *Information and Computation*, 98(2):142–170, June 1992.

[4] D. Drusinsky. The Temporal Rover and the ATG Rover. In *SPIN Model Checking and Software Verification*, volume 1885, pages 323–330. Springer, 2000.

[5] Paul H. Elliott. An Efficient Projected Minimal Conflict Generator for Projected Prime Implicate and Implicant Generation. Master's thesis, Massachusetts Institute of Technology, Feb 2004.

[6] D. Harel. Statecharts: A Visual Formulation for Complex Systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[7] K. Havelund, M. Lowry, S. Park, C. Pecheur, J. Penix, W. Visser, and J. White. Formal Analysis of the Remote Agent Before and After Flight. In *Proceedings*

of the 5th NASA Langley Formal Methods Workshop, Williamsburg, VA, June 2000.

[8] K. Havelund and G. Rosu. Monitoring Java Programs with Java PathExplorer. In *Proceedings of First Workshop on Runtime Verification, RV'01*, volume 55, Paris, France, July 2001. Elsevier Science.

[9] G. J. Holzmann. *The Design and Validation of Computer Protocols.* Prentice Hall, 1991.

[10] Michel D. Ingham. Mode Estimation Algorithms. Feb 6, 2003.

[11] Michel D. Ingham. *Timed Model-based Programming: Executable Specifications for Robust Mission-Critical Sequences.* PhD thesis, Massachusetts Institute of Technology, May 2003.

[12] A. E. Lindsey and Charles Pecheur. Simulation-Based Verification of Livingstone Applications. In *Proceedings of Workshop on Model-Checking for Dependable Software-Intensive Systems (DSN 2003)*, San Francisco, CA, June 2003.

[13] K. L. McMillan. *Symbolic Model Checking - an Approach to the State Explosion Problem.* PhD thesis, Carnegie Mellon University, 1992.

[14] N. Muscettola, P. P. Nayak, B. Pell, and B. Williams. Remote Agent: To Boldly Go Where No AI System Has Gone Before. *Artificial Intelligence*, 103(1-2):5–48, August 1998.

[15] C. Pecheur and R. Simmons. From Livingstone to SMV: Formal Verification of Autonomous Spacecrafts. In *Proceedings of the First Goddard Workshop on Formal Approaches to Agent-Based Systems (FAABS)*, Greenbelt, MD, April 2000.

[16] G. Gordon Schulmeyer and Garth R. Mackenzie. *Verification & Validation of Modern Software-Intensive Systems*, chapter 1. Prentice Hall, 2000.

[17] PolySpace Technologies. PolySpace Technologies - Automatic Run-Time Error Detection. http://www.polyspace.com, 2004.

[18] W. Visser, K. Havelund, G. Brat, and S.-J. Park. Model Checking Programs. In *Proceedings of the 15th IEEE International Conference on Automated Software Engineering*, Grenoble, France, September 2000.

[19] R. Washington, K. Golden, and J. Bresina. Plan Execution, Monitoring, and Adaptation for Planetary Rovers. *Electronic Transactions on Artificial Intelligence*, 4(A):3–21, 2000.

[20] Brian C. Williams, Michel D. Ingham, Seung H. Chung, and Paul H. Elliott. Model-based Programming of Intelligent Embedded Systems and Robotic Space Explorers. *IEEE*, 9(1):212–237, Jan 2003.

[21] Brian C. Williams and P. P. Nayak. A Model-based Approach to Reactive Self-Configuring Systems. In *Proceedings of AAAI-96*, 1996.