Composition and Correctness of Heterogeneous Planning Systems

by

Nicholas Pascucci

B.A. Computer Science, Colorado College, 2012

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

Master of Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2019

Author . . . .
Department of Aeronautics and Astronautics
May 23, 2019

Certified by . . . .
Brian C. Williams
Professor of Aeronautics and Astronautics
Thesis Supervisor

Accepted by . . . .
Sertac Karaman
Associate Professor of Aeronautics and Astronautics
Chair, Graduate Program Committee

Composition and Correctness

of Heterogeneous Planning Systems

by

Nicholas Pascucci

Submitted to the Department of Aeronautics and Astronautics
on May 23, 2019, in partial fulfillment of the
requirements for the degree of
Master of Science

Abstract

Autonomous systems present many new opportunities, especially for exploration in hazardous environments. One technique for building increasingly capable planning systems is to compose existing planners to enable specialization and division of subproblems. These systems require new analysis techniques, appropriate for ensembles of planners, if they are to be trusted with safety- and mission-critical roles in the future. Current state-of-the-art techniques address parts of this problem—including analysis of middlewares such as ROS and complex control systems—but have not yet provided analysis methods to address the particular correctness needs of composite planning systems. Applying formal methods to model the internal communications of planning architectures is a promising way to address this gap.

In this thesis, I develop a formal modeling method which enables proofs of correctness for planning system architectures which use a rich common data structure for both their inputs and outputs. The method is demonstrated through a case study of *Enterprise*, a system of planners developed at the MIT Model-Based Embedded and Robotic Systems (MERS) group which communicate using the Qualitative State Plan (QSP). The verification requirements of this system inform the development of a formal semantics for first order logic, defined in terms of the common data structure, which is useful for modeling systems of planners. Sentences in this logic can be used to express formal specifications about a planner's behavior, including correctness properties which are important for autonomous operations of critical systems. Using the logic one can also describe systems of planners built around this common data structure. Modeling of the *Enterprise* architecture and components in the case study demonstrates the usefulness of the technique.

The analysis method allows varying the level of abstraction by permitting the assumption of certain component behaviors by the architect. This allows the analysis to treat planners as "black-box" implementations while describing the rest of the system. Systems of planners can be described using specification composition, which enables description of various architectures. The use of intuitionistic mathematics enables mechanization of the logic in a variety of computer proof assistants to enable machine-checked proofs and implementation of planners by refinement from specification. Mechanization and opportunities to extend the method to more expressive logics are discussed as future work.

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

Acknowledgments

Defining, researching, and writing a thesis is not an easy task in the best of times but it is made more so by the love and support of those around me. I want to thank all of those who helped me get to where I am today, and those who helped me create the work you are about to read.

To my lab-mates at the Model-Based Embedded and Robotic Systems group: thank you for your support, insight, and feedback on a project which did not quite fit the mold, and for putting up with my abstract nonsense. Thanks to the Scouts group for being a great team and bringing me on many adventures.

To my girlfriend, Lauren: thank you for supporting me, understanding me, and always being there; for rubber-ducking and talking to me about abstract nonsense over scotch.

To Ethan: thank you for sparking my interest in category theory and formal methods and for teaching me so much about it, in addition to being a supportive and caring friend.

To all of my friends and family: thank you for helping me grow and become who I am today. Without you I wouldn't be here.

And finally, to Brian: thank you for giving me the chance to explore an area outside the norm for the group, and for your advice along the way; I hope that this work opens new horizons, as mine have been opened.

This research was funded by a grant from Exxon Mobil Research and Engineering Corporation through the MIT Energy Initiative, Award 023730-00025.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Robotic systems are becoming increasingly capable as technology improves. Advances in miniaturization have provided raw computational horsepower, allowing each new system to incorporate greater levels of artificial intelligence (AI) than its predecessors. In turn, these more capable robots are tasked with more and more difficult missions. Among these diverse applications of AI, space exploration presents both unique opportunities and unique challenges. Robots have cemented their role as the key agents for human space exploration in part due to their robustness, expendability, and low cost relative to living explorers. As our abilities to explore the solar system grow our machines are at the vanguard, venturing into new and more hazardous destinations. To succeed at the challenging missions we have created for them they will need to operate independently and autonomously, and to respond to unforeseen challenges while still achieving their mission objectives.

How best to achieve a high level of autonomy remains an area of active research, with model-based techniques presenting a promising approach. Provided with logical representations of their world, themselves, and their goals, these agents are able to reason using these models to determine the best course of action available to them. Model-based methods formed the basis of the first artificial intelligence system to control a spacecraft independently: the Remote Agent Experiment (RAX), launched aboard the Deep Space 1 spacecraft in 1999 [1]. As part of NASA's New Millennium Program (NMP), Remote Agent was intended to validate the use of AI to enhance system survivability and increase the likelihood of mission success. Although the project achieved its validation objectives [2], the vision of routine, frequent operation of fully autonomous, self-operating spacecraft espoused by the Remote Agent team [3] has yet to come to fruition.

A handful of successor projects have incorporated advanced autonomy technology into NASA spacecraft since, but none have reached the level of independence that proponents of the New Millennium Program had anticipated. Despite the NMP motto of "faster, better, cheaper," [4] NASA spacecraft remain costly to develop in both time and money. The resulting conservatism with regard to new technologies requires that only those deemed either low-risk or absolutely necessary for mission success are flown. New technologies are able to make inroads when the mission parameters dictate their use—see, for example, the limited automatic scheduling capability of the James Webb Space Telescope [5]—but overall the use of onboard autonomous systems remains rare despite their potential benefits. This is due in part to concerns about how to integrate these technologies into existing operational and technological structures, and in part due to a lack of trust in autonomous systems.

Gaining trust in AI systems is difficult, largely due to the unfortunate truth that increased capability generally comes at the cost of increased complexity. We require the systems playing a leading role within complex and expensive spacecraft to meet a high bar of quality and reliability. Standard flight control software deployed in a safety- or mission-critical role must meet certification requirements vouchsafing its suitability for the job. One might expect autonomous systems to be certifiable under these same guidelines, but due to their adaptability they are even more complex than "standard" software and testing their correct function in more than a trifling portion of the possible states the program may enter is next to impossible [6]. To provide a certification of correct system function suitable for mission critical software, more powerful techniques must be used.

Formal methods provide a possible way forward. By describing systems at a higher level of abstraction, formal analysis techniques are able to cope with the increased complexity more gracefully. They also provide stronger guarantees of correct behavior in the form of logical proofs that systems satisfy their specifications. Recent developments in the use of "lightweight formal methods" [7] have shown that the application of rigorous modeling and analysis techniques during the design stage is a useful tool for identifying conceptual errors at a reduced cost, and identifies bugs that testing does not.

In this thesis I demonstrate a first-order logical language aimed at this conceptual-design niche. The logic is suitable for writing specifications of correct behavior for AI systems built by composing several heterogeneous planners, which I term "composite planning architectures." This language focuses on semantic-level modeling of the communications between planners, as opposed to verification of the internals of the planners themselves or modeling of the communication channel. It is defined using a constructive mathematical basis that makes it suitable for use in computer proof assistants and as the starting point for correct-by-construction implementations using refinement techniques. Combined with an architectural modeling technique that can capture architectures at a variety of abstraction levels, we have the beginnings of a powerful tool set for designing and verifying composite systems.

To help motivate the development of the logical foundations of this language, I will provide examples throughout the thesis grounded in the *Enterprise* planning architecture developed by the Model-Based Embedded and Robotic Systems Lab (MERS) at MIT. *Enterprise* is itself a composite planning architecture, with a number of activity and path planners communicating to solve planning problems cooperatively. These planners use a common problem format, the Qualitative State Plan (QSP), that is both flexible enough to describe broad classes of planning problems and useful as the basis for planner specifications.

In the following chapters I develop the logical framework needed to provide semantics for first-order logic in terms of any data structure with the correct algebraic properties. I characterize QSPs and the planners operating on them, demonstrate that such a logic exists for QSPs, and how it can be used to perform analyses of the composite planning system at an architectural level.

To set the stage for these developments, we will first need to set a common modeling paradigm and lexicon. The next section provides definitions of the terminology used in the thesis, including *architecture*, *component*, and *specification*; a brief discussion of the modeling approach; and the formal problem statement guiding the research.

1.2 Approach and Innovations

Providing a useful tool for answering the specification problem involves modeling different aspects of the system in different ways. From a structural perspective, we require models of system components at a level of detail that enables proofs of their behavior, given the components they depend on, to be constructed. These proofs in turn are used to support specifications of component behavior expressed in terms of the logical properties of the messages exchanged between components, including simple properties such as termination or correctness with regards to constraints, and complex ones such as a guarantee of least-commitment. Finally, at an architectural level we need to model the concretization process and show that it maintains the invariants established by the previous work.

Scaffolding for this process is provided by the semantics imposed by the structure of the messages exchanged between components. In following chapters I will examine what it means for a message type to have sufficient structure for these semantics to be defined. As an example of this approach we will examine the Qualitative State Plan (QSP) structure deeply, demonstrating that it can support a semantics for intuitionistic first-order logic by showing that the set of QSPs forms a Heyting algebra. I then show how sentences in the specification logic encode specific sets of these objects that are exchanged between planners and satisfy certain properties. These properties in turn are a useful basis for proofs of correctness of the planners themselves as they provide both axioms and goals.

I also provide a number of worked examples demonstrating how to apply the logic to architectural problems, using *Enterprise* as a case study. Alongside formal descriptions of a common set of desirable properties, I present more detailed specifications of two planners: Sulu [8] and ScottyPath [9]. These planners are arranged into a number of example architectures whose properties I explore, leading to an example of how to use the logic for specification of novel systems. The logic is used during the design process to either support the safe reuse of existing planners or to create new ones using the specifications as a basis for implementation using correct-by-construction refinement techniques.

1.3 Structure of the Text

The following chapters discuss the background of research against which this work was done, the details of the specification logic, and a case study for the *Enterprise* composite planning system. Chapter 2 discusses related work with a focus on the analysis of complex spacecraft autonomy systems and existing formal approaches to modeling and verifying them. Chapter 5 describes in detail the mathematical foundations of the verification logic, with a case study examining the Qualitative State Plan structure used by *Enterprise*. Chapter 4 describes *Enterprise* in greater depth, motivating its design through real-world research deployments in oceanography. Chapter 6 then discusses the intricacies of modeling and a case study using *Enterprise*. Finally, Chapter 7 concludes the thesis with final thoughts and possible areas for future research.

Chapter 2

Related Work

The body of knowledge regarding analysis of autonomous systems is growing rapidly, as intelligent systems penetrate daily life and more attention is paid to certifying that these systems behave appropriately. A variety of approaches have been taken, including model-based and formal methods; however, the literature to date has focused primarily on either component-level analyses or concurrency problems and not sufficiently addressed the semantics of the internal communications of autonomy architectures in a formal way. The limitations of previous approaches to analysis and testing of autonomous systems, primarily the complexity of these systems and limited abstraction, are being eroded by recent developments in lightweight formal methods. In this chapter I describe the state of the research as of this writing, and point to gaps in the field that this thesis begins to fill.

2.1 Testing Autonomous Systems

Artificial Intelligence systems are beginning to permeate safety-critical systems across a number of terrestrial industries. Computer-controlled cars, planes, and factories are on the forefront of autonomy adoption and valuable lessons have been learned from their efforts—both successful and not. A key challenge encountered by all of these systems is that formulating good acceptance criteria for autonomous systems is quite difficult, due in part to the inherent complexity of software systems and in part to the necessity of modeling the operational context in determining the behavior of an adaptive system.

Software is much more difficult to validate than hardware. Unlike mechanical systems, software is able to take on a practically limitless number of possible states. Exhaustive testing of the state space is impossible for most software systems due to time and space constraints [10] [11]. Additionally, the behavior of the system in each of these states can be quite difficult to analyze as, due to the discrete mathematical nature of software systems, many traditional (i.e. continuous) engineering analysis tools do not apply [12]. For model-based AI agents, this difficulty is compounded because these systems operate at two levels: they contain models of the agent and its environment, along with a reasoning engine operating on those models. Models require different validation approaches from engines and have different lifecycles [13], with the engine being reusable across multiple models and models being tied closely to the system under control [14]. Because of this, investments of time and resources into validation and verification of the engines have larger payoffs over multiple missions if the engine can be reused [2].

In aerospace, these verification and validation (V&V) tasks would generally fall under the purview of a project's systems engineering team. To cope with the explosion in system complexity brought on by increased hardware and software capabilities, the systems engineering community has recently turned its attention towards their own model-based techniques known as Model-Based Systems Engineering (MBSE) [15]. MBSE refers to a broad set of approaches that emphasize capturing system designs and behaviors as a set of computer models, enabling systems engineers to simulate the system and identify problems early in the design cycle. Building the process around computable models, proponents claim, helps to expose assumptions and prevent divergence of understanding between the different parties involved in designing a complex system [11].

One of the existing MBSE approaches for designing spacecraft control systems is the State Analysis/Mission Data System (MDS) framework designed by researchers at the Jet Propulsion Laboratory [16]. The framework combines a modeling and analysis method (State Analysis) with a set of off-the-shelf software components. These pieces work synergistically to enable engineers to design sophisticated control systems by composing well-understood parts. At its core, State Analysis/MDS is a mechanism for identifying the possible states a system under control can be in and the possible control interfaces that can affect that state, so that the system designer can create a control system. The uniform approach allows techniques like MDS to provide stronger analysis results than previously have been achievable, while operating within finite resource constraints. However, MDS does not provide guarantees on the system behavior. Making models explicit within a system architecture is a necessary step towards full verification, but it is not sufficient; in the next section, I discuss formal approaches to verification that can provide true “for-all” guarantees.

2.2 Applications of Formal Methods to Autonomous Systems

Taking these techniques a step further, one might consider other types of analyses that can be performed on a system model. As I have already discussed, simulation and testing of a software system is not sufficient to determine correctness due to the large number of states it can be in. To get more leverage in analyzing large systems it is necessary to abstract the system and consider many possible states at once. This is the domain of formal methods.

At a high level, there are two major approaches to the formal analysis of software systems: model checking and theorem proving. Model checking is a fully automatic process that enumerates the possible state trajectories of a system model and attempts to discover paths leading to violations of a user-provided constraint on the system behavior. Current hybrid model checkers such as Alloy [7] are able to examine extremely large state spaces quickly, but a fundamental limitation of the technique is that it is restricted to a finite set of states. In many cases this may be “good enough”, and a growing body of work in *lightweight* formal methods encourages the use of model checking systems as a way to discover design errors earlier in the development process [17]. For safety-critical applications, however, the additional guarantees provided by theorem proving may be necessary.

Theorem proving relies on traditional mathematical techniques to produce proofs of correct behavior for a software system within a sound logical framework. Modern developments are generally supported by computer proof assistants such as Isabelle, PVS, Spark, or Coq, that implement consistency checking routines and programming languages for expressing proofs. Some systems, such as Coq and SpecWare, are designed to support the extraction of verified routines from the proof system itself (c.f. [18], [19]). By exploiting more powerful mathematical abstractions, proof-based techniques are able to tackle larger state spaces than model checkers. However, they require a deeper knowledge of both the system to be analyzed and the mathematics used to model it than model checking, leading to a higher barrier to entry for these systems and lower adoption among engineering teams [20].

Both techniques have been applied, with mixed results, to the verification of spacecraft software. During the course of the Remote Agent Experiment a concurrent effort was made to develop formal models of the system and analyze them for errors. This analysis was performed twice: once during the system development, and once following the identification of a concurrency error during the flight. In both instances the use of model-checking tools (in the first instance SPIN [21], in the second Java Pathfinder [22]) identified concurrency errors, including the flight error. However, these efforts were experimental and required the team to dedicate a large amount of time to modeling the system, with a particular emphasis on reducing the program state space to make model checking tractable. Most of the validation of Remote Agent was performed using scenario based testing instead of formal methods, though in the final *Remote Agent DS1 Technology Validation Report* the team stated:

> The primary lesson is that the basic system must be thoroughly validated with a comprehensive test plan as well as formal methods, where appropriate, prior to model development and flight insertion. ...
>
> The RAX experience confirms that testing FSW is hard. The bug that was found during flight shows that more attention and effort needs to be spent validating the basic engines. The validation cost is well worth the effort because the engines are components that can be re-used over many missions. [2]

The view that formal methods are useful but should be focused on slow-changing areas of the system, is a valid one. It indicates that to get the most benefit from formal modeling, the areas of the design that combine highest technical risk and a slow pace of evolution should be prioritized. Towards that end, the recent pushes in formal modeling have focused on the early design phases where mental models are not yet well developed, and where decisions affect large parts of the system and generally remain in place for the system's lifetime. This is the premise behind *Alloy* [17] and *TLA+* [23], which both employ a form of model checking to verify lightweight specifications but do not attempt to directly verify a program's implementation.

An alternative approach is to push the verification further back in the development cycle and embed a runtime monitoring system that can check the execution of the program against a specification. Because the system only needs to examine a single run of the program, the specifications are generally easier to write, and when combined with a provably-correct recovery system gives more confidence that the system will not enter erroneous states in an unrecoverable manner. These techniques can also be combined with testing to perform verification in the design loop [24], though they do not provide the same design guidance as the other techniques mentioned previously because they are performed *a posteriori*.

Ultimately the difficulty in using formal methods lies in how best to use abstractions to reduce the scope of the analysis. Design-time approaches deal with larger blocks of the system where implementation details are ignored. Runtime monitoring focuses on a single execution, rather than all possible executions. Both of these approaches are helpful, because they exploit modularity properties of the system to reduce complexity.

To be useful for analysis of large, complex systems, formal modeling techniques need to be made modular. Mixed-and-matched modeling techniques, applied where appropriate at various levels of abstraction, likely present the best balance between automation and confidence that the system contains no errors [20]. Within the robotics community both model checking and theorem proving have been applied to a number of verification domains ranging from probabilistic validation of system models [13] to analysis of middleware channels in ROS-based architectures [25]. Whether the problem can be formulated in such a way that a given technique can be applied is a major factor in determining the tools to use.

In the following section, I describe a particular domain that has been under-examined and will be the focus of the analyses in this thesis: semantic-level modeling of communications between planners, as opposed to verification of the planners themselves or modeling of the communication channel properties. This approach enables the use of either theorem proving tools or model checkers for verification, and addresses the question of correctness from a new perspective that is synergistic with other analysis methods.

2.3 Verifying Internal Message Semantics of Autonomy Architectures

Given the finite resources available for V&V, it is important to consider carefully where to apply them in order to get the greatest benefit. The current trend in robotics software design is towards component-based architectures where systems are built by connecting a set of reasoning components together using a middleware system such as the Robot Operating System (ROS). The majority of the verification done on this type of system focuses either on the components or on the middleware. Component verification involves showing that a given piece of software is compliant with a specification—for example, that a planner will not violate a risk bound by design [26] or that a planning system is free of runtime errors [14]. Analysis of the middleware linking these components tends to focus on the concurrency properties of the communications channel itself such as messaging latency, queue behaviors, and timeouts [25].

These approaches, either alone or combined, are not sufficient to fully characterize the behavior of the system. This is due in part because they are performed at a level of abstraction similar to that of the system itself. As Leveson argues in [10], to understand a system holistically one must take a higher-level perspective of it—one that both encompasses the system and enables reasoning about it using tools not present in the system itself. This is primarily because interactions between components are the driving factor of system-level behavior, and reasoning about these interactions holistically requires looking at the system from the outside. Emergent behaviors, those which arise from the interactions between components in a given system rather than from the individual components in isolation, are invisible unless the system is examined in this way [10].

One implication of this line of reasoning is that there is no singular approach to formalize this type of meta-analysis that applies in all situations. The existing analysis methods are incomplete; among the missing pieces, the robotics community lacks a systematic method for analyzing the communications that occur between components at a semantic level—i.e. what do the messages mean, within the problem domain—as a distinct perspective from looking at the properties of the communications channels carrying those messages. This perspective is desirable because while failures in the communications channel may ripple out through the architecture, semantic mismatches in the inter-component communications can have direct effects on system behavior even in the absence of component failure [10]: consider the scenario where a high-level planner dispatches sub-problems to child planners in a way that does not respect constraints between the sub-problems. Describing the communications flows at this level gives us another powerful viewpoint on systems of reasoning components that we did not have before.

This thesis describes an approach to connect the logical semantics of these messages to a symbolic language, enabling the system designer to write high-level specifications of planner behavior in terms of the messages exchanged with other planners. As a constructive mathematical approach, it is suitable for use in model checkers and theorem provers; and, with the help of a suitable tool set such as SpecWare, can enable the refinement and extraction of planner implementations from specifications [19]. Focusing on the logical properties of the messages is a novel approach to formal specification that has not yet taken hold in the planning community, and that can augment existing approaches to give a holistic view of the system under analysis.

2.4 Summary

Showing that a given software system is free of errors and satisfies its specification is a very challenging problem. AI systems make it even more difficult due to the very properties that make them useful, such as their adaptability and context-sensitivity. Model-based approaches are gaining adoption for the decomposition and analysis of complex systems, and for the task of verifying software the most appropriate models give formal properties that can be verified automatically. The current trend in the application of formal methods is towards earlier, design-time checking of high level system models and specifications.

Theorem proving and model checking are the two major approaches to this verification, each having different trade-offs; while model checking is easier to use for the non-specialist, theorem proving offers greater expressive power and abstraction enabling analysis of larger systems. Both of these techniques have been applied to robotics problems, largely in either demonstrating that components meet their specifications individually or that communications channels do not introduce system errors. Within the planning community, there does not yet exist an approach to modeling the communications between reasoning components that can describe the planning problems and solutions exchanged. The approach outlined in this thesis leverages the mathematical semantics of the data structures used to communicate this information between planners to enable architectures to be modeled precisely enough for model checking or theorem proving to be applied.

Chapter 3

Architectural Models & Problem Statement

In order to address the modeling problem formally, we will require a method for representing architectures in a way that makes them easy to analyze. This chapter outlines the modeling technique used throughout the thesis, and uses it to articulate the formal problem statement.

3.1 Notational Conventions

Throughout the text, I will refer to the following constructs using a common set of identifiers reflecting those used in this chapter.

- T is the generic type of the data structure used to express system models, mission goals, and planning solutions. When the type of an object may be ambiguous I will annotate types using the convention $x : U$, where x is the name of an object or variable and U is its type. The type $T \times T$ should be read as a pair of T's.

- A "fat" arrow, $\Longrightarrow$, represents implication within the "meta-logic" used to define specification logic terms.

- A "skinny" arrow, $\rightarrow$, has different interpretations depending on the context in which it used. In a type definition, such as for $f : X \rightarrow Y$, it represents the type of a function from type X to type Y. Otherwise, it represents implication within the specification logic.

- The object $m : T$ is a model of the system under control.
- The object $g : T$ is a specification of the goal state that the system must achieve.
- The object $p : T$ is the output of a planner.
- Planners are represented as functions of the type $T \times T \rightarrow T$, where the arguments are the model and goal specifications. In general I will use the symbol s to represent these functions. Function application is written as $s(m, g)$.
- The names of functions within equations and concrete implementations of T are typeset in Roman type (for example, $\mathrm{calls}(x, y)$ or $x : \mathrm{QSP}$). In text they are set in a monospaced font, such as `calls(x, y)`.

3.2 Problem Statement

The goal of this thesis is to show how to model software architectures so that they can be analyzed. The following section describes how these models are constructed and imbued with logical specifications in order to allow the designer to reason about their system at all stages of the design process from initial concept to final implementation.

Stated formally, the problem statement is as follows:

Definition 1 (Architectural Correctness Problem)**.** *Given a model of a software system comprised of an initial* architecture A, *a set of* concrete components C, *an* instantiation table I, *a set of* component specifications S_c, *and a* system specification S_s, *determine if the most concrete child architecture of* A *under* I *satisfies* S_s.

3.3 Modeling and Specifying Architectures

To describe a system's architecture formally, and to write specifications about it, we need a way to make a model of it. This model must capture the salient aspects of the architecture—namely, the components involved in the design and their relationships with each other—while retaining an appropriate level of abstraction. It should make analysis tractable and not unnecessarily restrict the possible implementations of the system.

All of the components for the architectures in this thesis will be planning systems. They interact by passing planning problems to each other in a synchronous, hierarchical calling style. There are several possible implementations of this architectural style, ranging from a single process that only uses its language's built-in procedure calling mechanisms, to a fully distributed planning architecture that uses network protocols to connect clusters of planning systems together. This modeling approach can be applied whenever the details of how the planners are connected can be safely ignored and only the semantics of the messages passed between planners are considered.

Once we have a model of the architecture, we can write specifications for how the pieces of the model must behave which are encoded in a suitable logic. In this section I explain how models are built, and how specifications are assigned to components in the model. Chapter 5 will provide a deeper treatment of the logic used to write these specifications.

Let us begin from the top, with an *architecture*.

Models of System Architectures and Components

Definition 2 (Architecture Model)**.** *An* architecture model *is a connected tree structure with directed edges, whose nodes are contained in a set of abstract components* $\mathtt{A}$ *and a set of concrete components* $\mathtt{C}$*. The edges are defined by a binary relation calls*$(\mathtt{a}, \mathtt{b})$*. When calls*$(\mathtt{a}, \mathtt{b})$ *is true, the component* $\mathtt{a}$ *is able to call on the component* $\mathtt{b}$ *to solve planning problems; diagrammatically this is represented with an arrow from* $\mathtt{a}$ *to* $\mathtt{b}$*.*

The component models represent both planners and systems of planners as mathematical functions. These functions are arranged by their calling relationships into a tree structure, as shown in figure 3-1. In this figure, we see a function `join_title` which calls two other functions: `concat` and `capitalize?`. In this example, we have a function for joining two strings together, and we know that we need a way to capitalize strings but we don't know how to do so yet. Sometimes we won't know which specific implementation of a component to use in a given part of the architecture and will need to defer the decision until later. A typical programming approach to this problem is to create a higher order function which can accept another function as an argument, but because the specification logic used in this thesis is first order, we will need another solution.

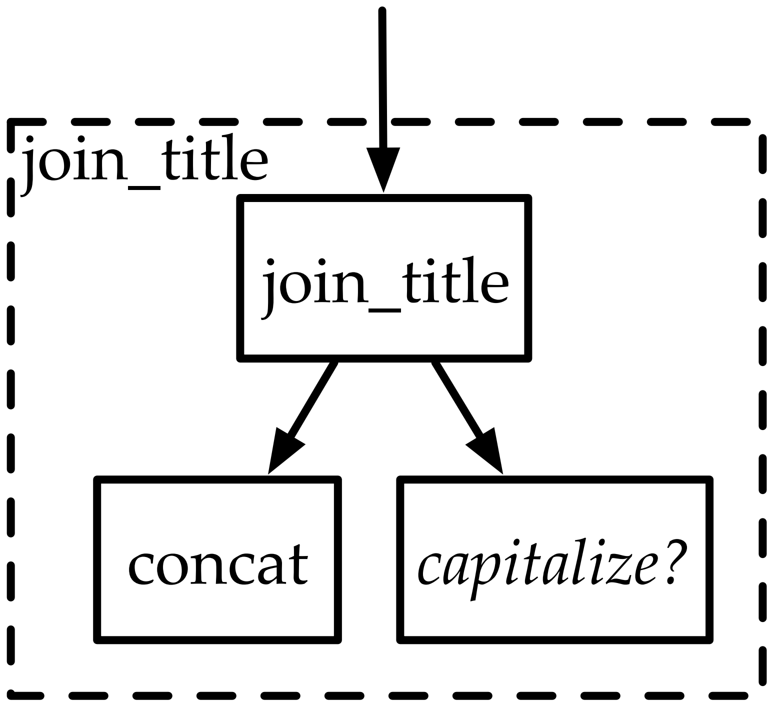

Figure 3-1: An example system diagram showing the functional tree structure created by an architecture.

Instead of allowing higher order functions which would make the logic more complex and difficult to analyze, the modeling framework allows parameterization of components through the use of an "abstract" component. These are named placeholders, like `capitalize?`, within the functional model of a component that can be bound to different "concrete" component. Binding a concrete component to an abstract component can be thought of in a similar way to the linking stage of compilation. This abstraction allows the structure of the system to be described independently of the components chosen to fill each role.

Definition 3 (Abstract Component). *An* abstract component *is an atomic identifier, unique within the architecture, that represents the use of another component whose implementation is not yet known. A given abstract component* c_a *may be used within the function definition of a component model* c_b *only when the relation* $\mathrm{calls}(c_b, c_a)$ *is true.*

Definition 4 (Concrete Component). *A* concrete component *is a model of a distinct software component. In this thesis these models represent planning systems and are expressed as functions of two arguments, having type* $T \times T \to T$*, where* T *is the type of the planning problem representation.*

Concrete components have a body *which defines the details of how its operational behavior is implemented.*

Component bodies are not explicitly represented in the architectural model, but instead are given in a separate model (c.f. Section 6.3). The component's body may refer to abstract components, but may not quantify over them or accept them as parameters in order to keep the specification logic first-order.

Note that the type given here for a concrete component differs from the typical one for a planner. A more common representation allows separate types for the initial state, plant model, environment model, and goal model. The approach used here unifies the initial state and goal state into a single state evolution objective, combines the plant model and environment model into a single object representing the invariant constraints of the system, and represents all of these pieces using the same type. This simplifies the mathematics considerably without sacrificing expressivity in the input representations. Subsection 5.3.4 justifies this choice in more detail; for now, we will defer this discussion and return our attention to the architectural modeling framework.

Making Abstract Architectures Concrete

We now have a mechanism for arranging concrete components, i.e. those we know how to implement, and abstract components, which we have not yet chosen an implementation for, into a hierarchical calling structure. Naturally we may find ourselves with architectures that have varying mixes of these two component models, ranging from entirely concrete models to entirely abstract ones. Usually a design will start with a very abstract model. As we discover more information during the design process we will want to be able to make abstract designs more concrete by binding the abstract components to concrete implementations. This process is called *concretization*, and uses an *instantiation table* to tell us which abstract components are bound to which concrete ones.

Definition 5 (Abstract and Concrete Architecture)**.** *When an architecture's set of abstract components* A *is non-empty the structure is called an* abstract architecture*; when* A *is empty, the architecture is a* concrete architecture*.*

Definition 6 (Instantiation Table)**.** *The* instantiation table *is a mapping from abstract components to concrete components, defining how each abstract component should be instantiated. The*

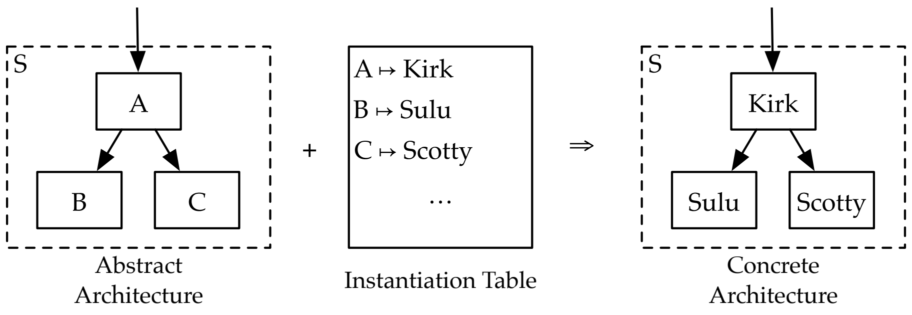

Figure 3-2: Concretization of an architecture using an instantiation table.

table must map every abstract component to at most one concrete component.

The process of converting abstract architectures to concrete ones with an instantiation table is illustrated in figure 3-2. Each abstract component is replaced with a concrete component in an iterative process, each step of which yields a *child architecture under the instantiation table*. When there are no remaining abstract components to replace, the child architecture is fully concrete.

If an abstract component does not have a matching concrete component in the instantiation table, the configuration cannot be fully concretized by that table; the resulting architecture is another abstract architecture that has been partially concretized. This might happen if an architect can't commit to a particular implementation of an abstract component—for example, it may not exist—but can make decisions about others. This flexibility is an important feature of the modeling technique, because it enables modeling to occur at all stages of the architecture's implementation, from conceptual design to final implementation.

Because the instantiation table maps each abstract component to at most one concrete component, we can see that the "most concrete" child architecture is the (unique) fixed point of the concretization operation under a given instantiation table, if it exists. When it does not exist the architecture cannot be instantiated; this may occur if there exist cycles within the dependency graph encoded by the instantiation table. This places a restriction on the types of architectures that can be modeled in this formulation.

Our definition of a model, and of the process of concretizing it, is restrictive in the sense that it does not permit circular calling patterns or heterogeneous subtrees that require binding the same symbol to different components. This is primarily a pragmatic consideration for the purposes of the thesis; more expressive formalisms are possible but require more structure from the specification logic.

Adding Specifications to Architectural Models

Our ultimate goal is to be able to construct a formal proof of correctness for the systems we are modeling. Beyond modeling the structure of an architecture and the components in it, we will also need to be able to attach logical specifications that restrict the behaviors of the components and of the overall system. These specifications will be written in a first order logic, and capture the properties of the components by describing the planning problems that they send to each other.

Definition 7 (Specification)**.** *A* component specification *is a first-order logical formula of the form* $\forall (m\, g : T),\ f(m, g) = x \rightarrow \Box$*, where* f *is the name of a component (planning over type* T*) that may be either abstract or concrete,* m *and* g *are the system model and goal specification to be planned for, and* $\Box$ *is a placeholder for a sentence in first-order logic defining the properties of the planner solution.*

Definition 8 (Specification Set)**.** *Several specifications can be combined into a* specification set*. Each specification may name a different component (*f *in the definition above), and restricts that components behavior.*

Generally an architecture will have a specification set which provides the desired behaviors of its components, and a concrete component will have a specification set that describes the actual properties imposed by its particular implementation.

The core of this thesis is dedicated to showing how the semantics of these specifications is defined. The particulars of the logic are discussed in greater detail in the following chapters. For the purposes of this section, it is sufficient to describe when a component "satisfies its specification under a given specification set".

Satisfaction of a specification set in this context means that we can demonstrate that when the component is used in a certain architecture it behaves correctly, assuming that the components it depends on also behave correctly. To do this we need to build a proof demonstrating that the truth of the sentence forming the specification of our component is a logical consequence of the component's implementation. The behaviors of the other components (as given by the specification set) are taken as assumptions or axioms for this proof. Reasoning from these axioms using a model of the component, we are able to decide whether or not the component will behave correctly in all situations. This process is described in more detail in Chapter 6.

Even if each component meets its own specification in isolation, it may not be the case that the system as a whole does. We must also demonstrate that the top-level component, the root of the call tree, satisfies its specification given a set of conforming child components.

We can say that an architecture satisfies a specification set if, for any choice of instantiation table whose concrete components individually satisfy their respective specifications in the set, the root component satisfies its specification. Because an architecture may be totally abstract or totally concrete, this definition gives us flexibility in modeling different structures and assessing the impact on the system's correctness when using different components.

Chapter 4

The Enterprise System Architecture

Before discussing a case study in applying the logic to a real system, it will be helpful to understand the origins and design of our exemplar. MIT's Model-based Embedded and Robotic Systems (MERS) group has iteratively improved on a common system for a number of years. This system, known as *Enterprise*, integrates multiple reasoning components within a unified framework to address planning and execution problems, with an eye towards spacecraft applications. In this chapter I will describe the principles guiding *Enterprise*'s development and discuss undersea exploration as an application of the system, setting the stage for the case study in Chapter 6.

4.1 Architectural Principles of Enterprise

In its original conceptual form *Enterprise* was described as a layered architecture as shown (figure 4-1), with human-facing components handling plan relaxation and negotiation at the top, a layer of executives and high-level planners in the middle, and a set of lower-level planners and diagnostics systems at the base [27]. This approach was straightforward conceptually, but in 2017 the implementation of the system had expanded to the point that it needed a renewed focus on the interfaces between planners. The existing approaches to integration tended toward tight coupling between components and difficulty incorporating new components as they became mature.

To give the system more flexibility, enable faster integration of new reasoning components, and simplify field deployments in 2018 the MERS group redesigned the system around a small number of well-defined application programming interfaces (APIs), as

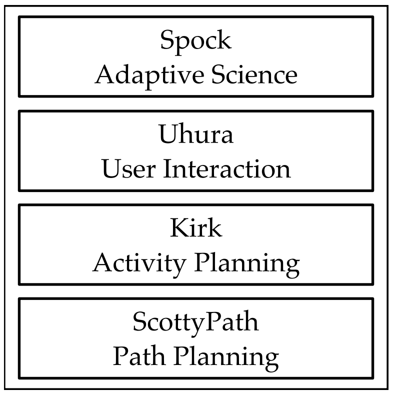

Figure 4-1: The original, simplified, Enterprise system architecture.

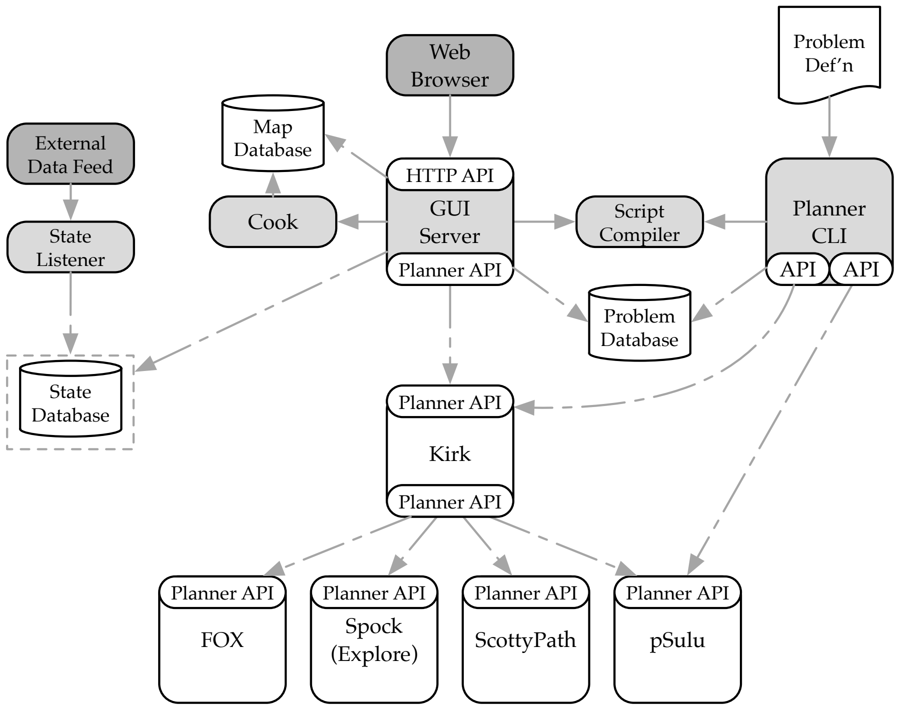

Figure 4-2: The revised system architecture, as of 2019.

shown in figure 4-2. Reasoning components meeting these specifications would be modular and able to serve a variety of roles both inside and outside a deployed *Enterprise* instance. Components could be easily substituted for each other, leading to faster testing and validation cycles and enabling researchers to take advantage of common infrastructure. User interfaces and other auxiliary systems would consume the same APIs as the rest of the system, giving them greater utility and a better return on the time and effort invested in building them.

Achieving this level of modularity required that we adopt a common representation for environment, action, and state models; that we provide a way to signal important properties of planning systems (e.g. risk awareness) in a composable way; and that we manage state within the system explicitly.

4.1.1 Common Representations

A common representation for planning problems, system models, and environments forms the communications fabric connecting all of the planners within *Enterprise*. Prior to the redesign effort various planners would consume and produce related structures in different encodings, making it difficult to integrate systems pairwise[1]. For example, the pSulu risk-aware path planner [26] used an in-memory problem representation implemented as a set of Lisp objects, while the ScottyPath continuous-time planner [9] read a Qualitative State Plan (QSP) represented in the YAML text format from disk. Our main activity planner, Kirk [28], acted as the architectural glue holding together the various reasoning components and performing impedance matching between their interfaces. An expected consequence of this approach is that Kirk quickly grew in complexity and became tightly coupled to the other planners.

To alleviate this problem we required a common representation to be used for planing problems, and for their solutions. For this purpose we adopted the Chance Constrained Qualitative State Plan (CC-QSP) that adds chance constraints[2] to the QSP structure discussed in Chapter 5. The CC-QSP was a good fit as it has several desirable properties: it is a superset of the most common plan representations in use among the *Enterprise* components; it allows expressing constraints on time, location, and risk; and it is easy to represent in a language-independent format. In addition, as I show in Chapter 5, it is possible to give precise semantics to the deterministic (i.e., non-chance constraint) portion of the CC-QSP, which aids in analyzing the system for correctness.

Models of the system under control and of the environment can, in principle, also be encoded as constraints within a Qualitative State Plan. In practice this is encoding is too

[1]Various standard problem formats have been used in *Enterprise* over time, including the Temporal Plan Network (TPN) and Reactive Model Based Programming Language (RMPL). As various planners advanced they implemented their own extensions and diverged from a common representation.

[2]Chance constraints are limits on the probability of constraint violation, given a stochastic model of the system under control. A good example of their use is to limit the chance that an underwater vehicle collides with terrain, given unknown currents and model errors.

"final" to provide the flexibility desired; instead, we use a more "initial" representation that encodes vehicle dynamics as Linear Time-Invariant systems within a matrix format and obstacles as GeoJSON geometry. These formats are also well defined, and the ability to model them as QSPs again simplifies analysis of the system.

While the CC-QSP is the current *de facto* plan representation, the system is able to support multiple formats in a standard way. This is enabled by the metadata system described in the next section, which gives each *Enterprise* component the ability to broadcast its own input and output formats and adapt to those of its peers.

4.1.2 Service Discovery and the Metadata Endpoint

To protect the modularity of the system it is necessary to provide a way for components to discover important run-time properties of their peers without having to encode specific knowledge about those peers ahead of time. For example, a component invoking a sub-planner may need to know if it can use the sub-planner to solve multi-agent problems. If it is able to discover the identity of its sub-planner it can gain this information, but this approach can quickly become brittle and prone to failure when another, previously unknown component is used instead. Our solution to this problem was to add a specific API call providing a *metadata* message that enables planners to inspect their peers.

The metadata message contained, among other data, the following fields:

- The planner's name;
- its capabilities, as a set of atomic, pre-determined symbols;
- supported file formats;
- a version identifier;
- and a recursive listing of the metadata of its own sub-planners.

This is a general-purpose method for planners to discover capabilities provided by their peers, and for other systems such as user interfaces to examine the system structure. Architectural flexibility is maintained by abstracting the planner capabilities into a set of common tag-like declarations. For example, pSulu is *risk-aware* while ScottyPath is a *multi-vehicle* planner; other planners can advertise these same abilities when they are installed in the system and be used in a similar fashion with no changes to the upstream components.

An important consequence of this method is that because each planner is in complete control of the capabilities it broadcasts at run-time, it may make decisions about what guarantees it can provide based on the currently-deployed system architecture. As a hypothetical example, a planner that can provide risk-aware planning when given a risk-aware sub-planner but not when given a naïve one may change the capability set it broadcasts based on what it is able to determine from its children's metadata. This enables flexible architectures, and incremental enhancements can be made locally without impacting the global structure.

4.1.3 State Management

When designing a composable system it is natural to imagine each component as a pure, stateless function, transforming inputs into outputs directly and with no side effects. While this model is conceptually quite simple and valuable for analysis purposes, it does not reflect the fact that real systems maintain state and use it for a variety of purposes, not the least of which is to improve their own performance. To preserve the major benefits of the functional approach while enabling components to use state effectively requires taking a principled path towards managing state.

As a motivating example of why state management matters, consider that several of the planners in *Enterprise* retain the state of their solvers after completing a plan in order to be able to reduce the time required for subsequent planning problems. This "warm start" capability is extremely useful in improving planner performance as it reuses knowledge of the problem domain captured in the solver state. At its most beneficial, this type of strategy can act as a memoizing cache and return pre-made solutions to problems that have already been solved. Unfortunately, the naïve approach to implementing this capability—namely, to keep the solver state in memory—carries a number of issues. When planners encounter unrecoverable errors or must be restarted for another reason (such as for a system upgrade) they lose their state. Caching systems are ineffective or potentially incorrect when they do not have access to the entire problem context, including the solver state. Performance benefits of reusing previous solutions are not realized when there are multiple replicas of a planner, and synchronizing their states is difficult.

To address these issues we introduce the concept of a *state token*. State tokens are an opaque, unique identifier that the planner provides to clients along with the results of a planning session in order to recreate the saved state of the solver[3]. They may be implemented in any number of ways, including as a direct encoding of the solver state in the token or as an obfuscated database key pointing to persisted state stored elsewhere. On subsequent requests the client may provide the token to the solver as a "hint" about how to best approach the problem, potentially realizing large performance gains with no compromises in correctness. In addition, because the state is now explicitly encoded in the request context, a number of new architectures are possible that enable the use of caching systems between planners, horizontally scaling planners through replication and load balancing, automatic failure and restart tolerance, and more.

One additional benefit of this approach is that it makes the full system quite easy to model by extending the naïve functional model mentioned at the beginning of this section to include the solver state as an input. By doing so, analysis of the correctness properties of the system is dramatically simplified, and reasoning about state becomes less burdensome. For the purposes of this thesis, we will make the additional assumption (justified by the properties of the planners in practice) that the final result of a planning session is invariant with respect to the state of the planner, and that the only property changing when state is changed is the speed with which that solution is produced.

4.2 Application: Undersea Exploration as a Space Analog

As described in earlier chapters, it is quite difficult to find opportunities to deploy state-of-the-art AI systems aboard spacecraft due to the high cost of such missions and the risk posed by novel software systems. To gain experience with real-world AI assisted missions, the MERS team has partnered with the Woods Hole Oceanographic Institution (WHOI) to deploy *Enterprise* on oceanographic research expeditions.

These expeditions are a particularly valuable space analog because they provide a challenging and often dangerous environment for robotic explorers, where loss of communication is commonplace. Changing tides and currents along with bathymetric obstacles introduce a number of risks for the vehicles including loss of power, collision with terrain,

[3]A state token is conceptually similar to the "cookies" used in web browsers to store session state.

and inability to complete planned trajectories within timing constraints. These challenges are a good match for *Enterprise*, which can apply multiple planning techniques to address them.

Space analogies extend to the concept of operations (CONOPS) as well. As part of the proposal for a series of joint MIT-WHOI missions under the auspices of the NASA Planetary Science and Technology from Analog Research (PSTAR) program, the CONOPS for a generic exploration mission was described as involving three primary system roles working in concert: an orbiter, a probe, and a lander. Upon reaching the research site, which may be an exoplanet in the space context or an area of the ocean in the Earth-bound context, the orbiter collects high-resolution mapping data for use in future mission planning and identifies areas of scientific interest. It then dispatches a probe to perform additional reconnaissance to determine if these sites are worth detailed study and to update predictive models used for identifying further visitation goals. Finally, a lander is dispatched to perform more extensive investigation of the highest-value areas. Each of these stages provides more information at a correspondingly higher cost. It makes sense, therefore, to try to quickly identify the best possible use of the most expensive asset (the lander) by targeted deployment of the remote sensor platforms (orbiters and probes).

In the oceanographic context we can draw analogs to these roles: the research vessel with its array of sonars, water column sensors, and surface-based instrumentation is the "orbiter." A team of low-cost, low-capability autonomous underwater vehicles (AUVs) are the "probes," and a high-cost but highly capable remotely operated vehicle (ROV) is the "lander." Maximizing the strengths of this joint system while minimizing the costs of operations benefits from powerful planning capabilities.

We have deployed *Enterprise* on a number of these expeditions, but I will focus primarily on the January 2018 mission to Hawaii and the December 2018 deployment to Costa Rica. It was in the time between these cruises that the MERS team converged on the modular architecture described above and began to reap the benefits.

4.2.1 Deployment: Au'Au' Channel, Hawai'i

In January of 2018 the MERS team was part of a multinational group of researchers aboard the research vessel *Falkor*, operated by the Schmidt Ocean Institute, whose objective was to explore the coordination challenges implicit in operating large numbers of heterogeneous

AUVs concurrently. Additionally, our scientific objectives also included identifying and capturing imagery of sea bottom coral formations within the Au'Au' channel between the islands of Maui and Lna'i. Operations during the cruise included the coordination of vehicles from the Woods Hole Oceanographic Institution (WHOI), the Australian Center for Field Robotics (ACFR), the University of Rhode Island (URI), and the University of Michigan (UM).

A typical day during the cruise would involve committing to a high-level mission plan developed during the previous evening, which outlined particular deployment zones and asset allocations (e.g. the availability of small boats and *Falkor* crew to recover vehicles) to service the science objectives. During the course of the day various vehicles would be deployed and recovered, sometimes multiple times, with groups operating in parallel. Plans evolved quickly in response to new findings, hardware or software failures, resource contention, and other factors, making planning and coordination of the research groups a continuous challenge. Automated planning tools with the ability to de-conflict various teams and perform negotiations to resolve over-constrained problems were in high demand.

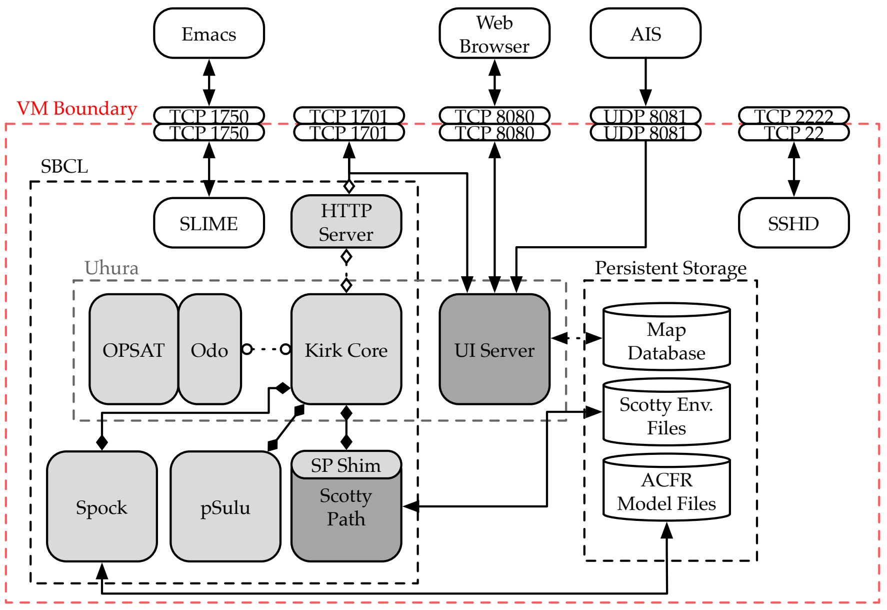

Figure 4-3: A block diagram showing the virtual machine architecture used in the Hawai'i deployment.

The *Enterprise* system deployed by MIT was used to produce risk-aware trajectory plans for all of these various teams over the course of the cruise, and through the process of designing models and planning missions we discovered that our existing architecture was neither flexible or robust enough for our needs. The system deployed in Hawai'i was built using a VirtualBox virtual machine with pre-installed software support for the various planner systems and run them within a contained environment. Figure 4-3 shows the conceptual architecture of this virtual machine. While this approach was portable and allowed us a consistent environment in which to run the planners, it was also brittle, complex, and difficult to maintain. Integration between the planners was *ad hoc* and the input formats poorly specified, leading to flakiness in the system and difficulty tracking down bugs. Because each system was implemented independently using different guiding principles, researchers familiar with one system could not expect to meaningfully contribute to others.

Our diagnosis of these issues directly informed the redesign of the system around the architectural principles described previously, and a migration to the Docker containerization technology as the basis for future deployments. We exercised these revised system during the following expedition, this time in Costa Rica.

4.2.2 Deployment: Puntarenas Province, Costa Rica

Our next deployment, to Costa Rica's western shores, demonstrated the flexibility and stability of the new design. Each of the *Enterprise* components was encapsulated into a separate Docker image, with the arrangement of the various pieces defined externally using the Docker Compose tool set. Containerization enforced communication to occur along explicit channels described in the system architecture, and standardization of these lines of communication enabled the containers to be connected in an ad-hoc manner to create new Enterprise instances.

The decoupling enabled much faster research iterations. Now, instead of having to rebuild a virtual machine containing the entire system at once when a change was made—a slow, fragile process that could take hours—each component could be built independently with a clean and deterministic process, yielding predictable results. New features could be developed, built, and deployed within an hour without incurring system downtime. We exploited these properties to create continuous integration and continuous deployment (CI/CD) infrastructure that would automatically compile our software components and deploy it to the *Falkor*'s computing infrastructure when new changes were made. This reduced the amount of overhead involved in deploying the system dramatically.

With a more understandable, flexible, and stable system the team was able to conduct research rapidly and focus their time on collaboration with the other members of the science crew. The enhanced focus on research rather than engineering meant that our scientific results improved: our system was able to successfully combine an information-seeking path planner (Spock [29]) and an activity planner (Kirk [28]) to plan mission trajectories for both an AUV (a Teledyne-Webb Slocum Glider) and a remotely operated vehicle (the SOI ROV "Subastian") to identify a number of undersea hydrothermal seeps. At the project level, our composable architecture helped to keep us insulated from failures and programmatic risks. Components that encountered difficulties in development and were unable to be deployed would have been blocking factors previously but were easily omitted in the more modular design.

4.2.3 Future Developments: Santorini and Beyond

The next deployment in this series will occur in 2019 at the Kolumbo undersea volcano in Santorini, Greece. This site is home to a number of undersea carbon dioxide pools, presents a hazardous environment that is challenging for robotic operations, and may host unique lifeforms that have yet to be discovered [30]. Within the caldera, strong currents that change direction quickly and the limited maneuvering space present particular challenges for thrust-limited vehicles such as Slocum Gliders. In contrast to the three-week timeframes of previous missions, our excursion to Santorini may be compressed to as little as one week. To address these challenges, *Enterprise* will need to be extended with new capabilities.

Modularity of the system architecture is the key enabling factor for this extension. We are currently developing new system components that exploit the work previously done to standardize the system. A new system running on-board the gliders will enable a microcosm of the larger Enterprise system to be used for real-time control of the vehicle, giving us more robust maneuvering options and the ability to replan missions in response to events detected while underwater. Ship-side systems will help the principal investigator manage the campaign, providing stochastic scheduling capabilities and resource management tools to aid in replanning and more seamless adaptation to contingencies.

Beyond our oceanographic deployments, we are taking advantage of the recent proliferation of industrial cloud computing services to host a cloud-based *Enterprise* system for instruction and research. Our containerized planners are easily deployed using the Kubernetes orchestration tools to services such as Google Cloud Engine. Explicit state management and common interfaces enable us to create generic load-balancing and caching systems that abstract the individual planning systems and allow us to scale horizontally. We intend to create a system to provide "planning as a service" for use at MIT and other institutions, with the intention of enabling more researchers in academia and industry to leverage *Enterprise* for their projects.

4.3 Correctness, Validation, and Verification in the Modularized Architecture

A plug-and-play architecture has many benefits, as described above; however, it also brings a number of challenges. The architecture must simultaneously provide enough expressive power to enable the components to be used to their fullest extent while restricting the communications between components to ensure their interoperability. Unforeseen interactions at the system level can cause difficult to diagnose failures. Ensuring that the output of a composite planning system, built in this modular fashion, is correct requires reasoning abstractly about the space of possible components and their properties. The next chapter will apply the modeling and specification techniques developed so far to the *Enterprise* architecture, demonstrating the approach in the context of a real-world example.

Chapter 5

Logical Foundations

5.1 Introduction

We can now turn our attention to the logical structures we will use to write the specifications for architectural models. This logic needs to be powerful enough to express interesting properties, have a well defined semantics to enable its use in proofs, and be possible to mechanize so that system designers can leverage computer assisted proof tools such as theorem provers or model checkers. In designing a logical language to meet these criteria, we will need to leverage the tools developed in the study of mathematical logic.

Over the course of this chapter we will examine the mathematics that underlie the interpretation of first-order logic using algebraic structures, and develop an understanding of the properties required for this to be done successfully. As a case study in this type of modeling, we will then examine Qualitative State Plans (QSPs) and demonstrate that they possess the requisite structure. The result of all of this work is to provide a concise and expressive language for describing the properties of planning systems.

We will begin with a review of the fundamentals of intuitionistic logic and key operations on lattices needed to define its semantics in set-valued structures. This groundwork will lead to the definition of a *Heyting algebra* and of the axioms such a structure must satisfy. Next we will examine Simple Temporal Constraint Networks [31] with attention to their lattice structure. After describing state constraints, we proceed naturally to the Qualitative State Plan itself and extend the model to include state.

5.1.1 A note on notation

When defining a logic confusion occurs easily when the context of operators is unclear. For this thesis, I use the following conventions:

- A "fat" arrow, $\implies$, represents implication within the meta-logic used to define specification logic terms.
- A "skinny" arrow, $\rightarrow$, represents implication within the specification logic itself.
- The standard symbols $\wedge$ and $\vee$ for logical "and" and "or" are given subscripts when used to represent specification logic terms, if their interpretation is non-obvious given the context.

5.2 Fundamentals of Intuitionistic First-Order Logic

There are two major schools of thought with regards to mathematical logic that dictate how semantics are defined and what can be proven in a given language. The classical approach, which is the most common form of logic used in mathematics, has fundamental limitations when used for creating verified software.[1] In contrast, the intuitionistic approach, which is stricter and less commonly used for pure mathematics, gives more useful results for software in part because of its limitations.

This section begins by discussing the history and distinctions between these ways of thinking about logic, motivating the use of intuitionistic mathematics in defining the logic for writing specifications of our architectures. I will then discuss how a logical signature is constructed to define the logic's syntax, and show how the semantics of our first order logic can be defined in a clean, precise manner through the creation of a *Heyting algebra*, which is a general method us to create logics describing planning problems or other interchange formats having a certain structure. We will then apply these ideas to the *Qualitative State Plan (QSP)* structure used in the *Enterprise* system as a case study in defining such a logic for a real-world data structure.

[1]For example, it is quite possible to write a sound classical proof that cannot be feasibly implemented as an algorithm. We will see this type of limitation again later in the chapter.

5.2.1 Historical Background

Multiple formalizations of mathematical logic are available to the practitioner. Variations of classical logic have been the primary foundation and working language for mainstream mathematics since the time of Aristotle [32]. As Lindstrom, et al describe in their introduction to the 2009 work "Logicism, Intuitionism, and Formalism," the situation changed in the late 1800's with Frege's work *Begriffsschrift*, in which he tried to provide a sound, complete, and consistent logical basis for mathematics. Following publication there began a rapid expansion in the body of work re-examining the foundations of mathematics and logic, leading to the creation of three major schools in the field: *Logicism*, *Formalism*, and *Intuitionism*. These largely differ in whether they consider logic to be "fundamental," in the philosophical sense of existing on its own without needing to reference anything outside itself, in the way that a physical object might be said to simply "exist."

In *Begriffsschrift*, Frege argued that logic could be defined independently of mathematics, and then used as a system in which mathematics could be formalized. This mean that the logical system itself was designed on its own and accepted as consistent *a priori*, relying on its own internal mechanisms to maintain the truth of statements deduced using the logic. Axioms in this system were to be "self-evident," not requiring external justification to be accepted as true. This school became known as *Logicism*.

Formalists, in the mold of Hilbert, sought to identify consistent axiomatic systems for their own sake. In contrast to logicism which relied on the mathematician to determine what was true using the self-evidency test, for the formalists consistency was a sufficient condition for the logical objects described by the system to exist. Both of these programs to formalize mathematics were jeopardized by the work of Russell and Gödel [33], both of whom demonstrated that there are potentially crippling limitations to what can be determined using any logic which is powerful enough to be considered "useful" for the basis of mathematics.

Intuitionism, in contrast to logicism or formalism, began with Brouwer's rejection of what he considered the unconstrained use of the "law of the excluded middle" (LEM) ($\emptyset \vdash p \vee \neg p$). The LEM is generally accepted within classical logic and states that in every circumstance it is possible to prove that a statement is either true, or that it is false; if it is not one then it must be the other. The "middle" being excluded is the unknown state between truth and falsity, where the proposition p is neither true nor false.

Brouwer claimed that when used improperly the LEM would lead to unreliable deductions. Existence proofs in particular become suspect when they rely on proof by contradiction in ways that depend on the LEM, because it is possible to write a proof that an equation has a solution that cannot be used to construct any examples of structures that satisfy it. In this sense, the proof is not "information bearing" about the objects it describes [34] in that it does not show how to create the objects themselves.

For Brouwer mathematics was the free practice of exact thinking, where truth was constructed, rather than discovered, by the mathematician by applying a series of precise inference steps to arrive at what he termed a "mental construction." Mathematics, then, is a fundamental activity[2] of the human mind and not rooted in a separate entity (such as logic) as the logicists or formalists would claim. Rather, Brouwer insisted that mathematics was the basis of logic and not vice versa. In his view, unless a structure can be synthesized in some deterministic fashion there is no meaningful proof of its existence [35].

This property of intuitionism, that every proof be reducible to some algorithm, is a very useful one for anyone writing specifications about software systems because it restricts our methods of proof to the realm of computable functions. If we cannot algorithmically construct an object, it does not matter if it can be shown to exist in a classical sense. Specifications written in a classical logic, which places no such limits on whether non-information-bearing proofs are acceptable, leave themselves exposed to the case that there are no possible implementations of the specification. Using intuitionistic techniques instead means that every proof not only provides a logically sound deduction of a certain fact, but also embodies the computation necessary to construct an object having the proven properties. These objects are called *witnesses* to the proof.

Fortunately, if unintuitively, intuitionistic logic has been shown to be strictly more powerful than its classical counterpart. Any sentence written in classical logic can be rewritten into intuitionistic logic through the appropriate translation of the classical sentence into an intuitionistic version [36]. This process is called *Gödel Translation*, and gives a sentence in the intuitionistic logic that is true if and only if the original is true in classical

[2]A "fundamental object" or "fundamental activity" refers to a concept that can be considered the most basic or the most atomic in a philosophical sense, such that describing its existence is possible a priori, without relying on the existence of other objects.

logic.

In the intuitionistic view logic is simply a construction atop more fundamental objects and not a fundamental object in itself. As a result, logic should be given meaning in terms of mathematics and the properties of its structures. This means that to interpret a logical sentence such as $p \wedge q$, one needs to know what mathematical objects p and q are, and what operation on these objects $\wedge$ is. Without this grounding, the sentence doesn't really "mean" anything at all.

As a student of Brouwer's, Heyting formalized this idea by demonstrating that certain structures of mathematical objects were rich enough to be the basis for defining the semantics of certain logics [33]. We will specifically make use of his formalization of first order logic, the Heyting algebra, to define our own specification logic by using the properties of Qualitative State Plans.

5.2.2 Signatures of First Order Logics

To set the stage for the definitions to follow, we will first want to have a working definition of what a first order logic is, and what operations it supports. We will begin by defining a *signature*, which bundles the operations and objects of the logic together.

Basic propositional logics contain a set of *atomic objects* representing statements that have some truth value and cannot be broken down into smaller statements. These objects are then joined together to form compound statements using logical connectives, such as "or" and "and." More advanced logics can include functions which transform other statements, relations between statements, and other structures of varying complexity. We will build a first-order logic to use for specifications, as it provides enough power to write interesting specifications without adding undue complexity.

A *first-order* logic is a propositional logic that allows quantification over the atomic objects, but not over functions, relations, or compound formulae. The *signature* for a general first order logic contains a set of atomic objects P; two distinguished values of P written $\top$ (truth) and $\bot$ (falsity); the connectives $\vee$ (or), $\wedge$ (and), $\rightarrow$ (implies), and $\neg$ (negation); an infinite set of variable names V that may be substituted for propositions; a set of relation symbols R; a set of function symbols F; and the quantifiers $\forall$ (for all) and $\exists$ (there exists).

For the purposes of this thesis we will concern ourselves only with *single-sorted* logics;

these are the logics that have a single "type" of object in P, and all relations and functions act solely on this type. This is sufficient to model planning architectures which use a common data structure and avoids the need to model types explicitly within the logic. The inclusion of function and relation symbols is necessary to model planners (which we model as functions[3]) and equality (which we model as a relation).

We can combine these elements to create logical formulae, which may contain variables of type T. The *free variables* $FV(\phi)$ of a formula are the variables appearing in ϕ, drawn from V, that are not bound within the scope of a quantifier. Drawing from the lucid treatment of various logics by Johnstone [37], the formulae of a signature are created according to the following rules:

1. Every atomic proposition and every variable is a trivial formula. The free variables of an atomic proposition is the empty set; for a variable, it is the singleton set containing the variable itself.

2. $\top$ is a trivial formula as is $\bot$, neither of which contain any free variables.

3. If P and Q are formulae, then so are $P \vee Q$ and $P \wedge Q$, the free variables of which are given by $FV(P) \cup FV(Q)$.

4. If P is a formula, so is $\neg P$. The free variables are the same for both.

5. If P and Q are formulae then so is $P \to Q$. The free variables are given by $FV(P) \cup FV(Q)$.

6. If P is a formula, and x a variable, $\forall x, P$ and $\exists x, P$ is also a formula. The free variables in either case are given as $FV(P) \setminus \{x\}$.

7. If R is a relation or function symbol of arity n and $x_1, \ldots, x_n$ are formulae, then $R(x_1, x_2, \ldots, x_n)$ is also a formula whose free variables are the union of the free variables of the assorted x_i.

As in Johnstone [37], generally we will assume that the formula are chosen such that the names of their variables do not collide and that formulae differing only in the names of their variables are equivalent. This property is known as *alpha-equivalence*. Formulae

[3]The choice of how to interpret the function symbols is made by the instantiation table when the architecture is concretized; see the definition of *abstract component* in Chapter 3.

containing no free variables—that is, those that either contain no variables at all or all of whose variables are bound within the scope of some quantifier—are "closed formulae" while the remainder are "open." A *theory* over a signature is a subset of the signature's closed formulae that are considered axiomatic or self-apparent. When modeling architectures, we will often axiomatize the behavior of a planner's sub-planners to create a theory. Within this theory we can then form a proof that if the sub-planners behave correctly (that is, according to our assumptions) then the parent planner will also.

These signatures are considered "first order" because the possible scope of quantified variables is restricted to the elements of P. While function and relation symbols may be included as part of the signature, they cannot be bound by quantifiers and must appear as named constants within formulae containing them.

5.2.3 Heyting Algebras

Providing a mathematical semantics for first order logic requires defining an object in a mathematical structure corresponding to each formula generated by the logic's signature. A common example, and one which we will use to illustrate concepts in this chapter, would be a set containing particular elements. When we create a logic whose semantics are defined in terms of the theory of this set and its subsets, every formula will have a corresponding subset and the overarching structure is a lattice of these subsets.

These mathematical objects must also support operations corresponding to the logical operators we want to use. The two operations, mathematical and logical, need to mirror the ways their counterpart behaves. Continuing with our set example, in order to define the logical "or" operation there needs to be a binary operation $\vee_{set}$ on sets that we can use which behaves in the set-world the same way that $\vee$ behaves in the logic-world. If we have logical statements p and q and sets P and Q, then the set that corresponds to $p \vee q$ is always $P \vee_{set} Q$, and the same holds for all of the other operators.

The Dutch mathematician Arend Heyting provided the axioms and interpretations required to provide such a semantics for intuitionistic first order logic in his 1930 work "Die formalen Regeln der intuitionistischen Logik," resulting in a structure known as a *Heyting algebra*.[4] We will begin our examination by describing what each of these terms

[4]In a nutshell a Heyting algebra is "a bounded lattice, with all *meets* and *joins*, equipped with an implication operator satisfying certain axioms."

means and then proceed to discuss examples of Heyting algebras to help anchor the concept firmly. In defining the Heyting algebra we will see how to connect logical structures and mathematical structures so that we can move between them freely.

Fundamental Structures: Lattices, Joins, and Meets

Lattices are structures which can be built from sets that meet certain criteria: the objects in the must be partially ordered, and for every pair of objects in the set there must be another object, also in the set, which is their *meet*, and yet another that is their *join*. We will see that by finding these types of objects in the lattice we are able to interpret logical sentences, but first we need to understand what properties they have.

To begin with, the set must be equipped with a partial order relation[5] between elements, "less than or equal to" ($\leqslant$), that satisfies the following axioms:

1. Reflexivity: $\forall a, a \leqslant a$.
2. Antisymmetry: $\forall ab, a \leqslant b \wedge b \leqslant a \implies a = b$.
3. Transitivity: $\forall abc, a \leqslant b \wedge b \leqslant c \implies a \leqslant c$.

This ordering relation imposes a directed graph structure on the set, where there is an edge from an element *a* to an element *b* if and only if *a* is less than or equal to *b*. When this graph has"root" and "leaf" nodes which have no objects that are greater than them (or lesser than them, respectively) the lattice is "bounded". These objects are usually distinguished and named the "top" ($\top$) and "bottom" ($\bot$) nodes. These bounds must satisfy these sentences:

1. $\forall a, a \leqslant \top$.
2. $\forall b, \bot \leqslant b$.

Within the body of the lattice there are additional distinguished locations with respect to every pair of nodes: their *join* and their *meet*. Joins and meets are, fundamentally, ways to find the "best" way of combining two elements such that certain properties are preserved. A join of two elements *a* and *b* (also known as their "supremum") is the least

[5]Also referred to as a "poset," for "partially ordered set."

element that is larger than both *a* and *b*. We will write joins within a lattice l using a similar symbol as logical "or," $\vee_l$, for the precise reason that a join within a lattice structure is the semantic interpretation of the "or" operation in a corresponding logical language. Similarly, meets ("infimums"), written as $\wedge_l$, correspond to logical "and." The following axioms define these operations precisely:

1. Joins: $\forall abc, a \vee_l b = c \iff (a \leqslant c) \wedge (b \leqslant c) \wedge (\forall d, a \leqslant d \wedge b \leqslant d \implies c \leqslant d)$.

2. Meets: $\forall abc, a \wedge_l b = c \iff c \leqslant a \wedge c \leqslant b \wedge (\forall d, d \leqslant a \wedge d \leqslant b \implies d \leqslant c)$.

Note that, for the join or the meet to exist, there must be a unique element satisfying the above axioms. If a set is equipped with a partial order, and contains a join element and a meet element for every pair of elements, it is considered a *lattice*, and if it also has bounding elements as described above it is a *bounded lattice*. One common example of a lattice is a set and its subsets; we will use a small set throughout the chapter to help explain these concepts. An example of a structure which is a partially ordered set but not a lattice is the set $\{1, 2, 3\}$ ordered by divisibility (e.g. $1 \leqslant 2$, $2 \nleqslant 3$). This set does have a meet for every element, but it does not have a join for the elements 2 and 3.

To illustrate these concepts, let us examine a small lattice consisting of the subsets of $\{A, B, C\}$ ordered by subset inclusion. We will return to this example to illustrate the implication operation.

Within this lattice the bounding elements are $\{A, B, C\}$ at the top and the empty set $\emptyset$ at the bottom.

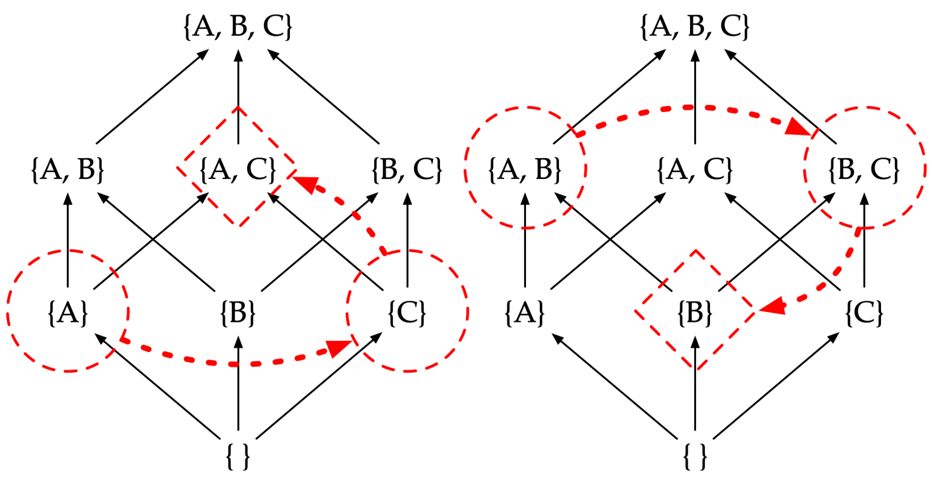

Figure 5-1: The join $\{A\} \vee \{C\} = \{A, C\}$ *and the meet* $\{A, B\} \wedge \{B, C\} = \{B\}$.

A join in this lattice is the union of two sets. We can verify this by seeing that for any two sets their union is the smallest set that is larger than both of them: it contains no extraneous elements, and contains all of the elements of the two "input" sets. Similarly a meet is the intersection of two sets, and, because it contains exactly the elements that are present in both, it is the largest set that is smaller than its inputs. An example for both joins and meets is shown in figure 5-1.

It should be clear how the two operations relate to their logical counterparts, *or* and *and*: meets are conjunctions, and joins are disjunctions. This is true in general, and gives us definitions for these two logical operators in any Heyting algebra. The analogs for truth and falsity are also apparent: they are the top and bottom elements of the lattice (that is, the maximal and minimal elements as given by the lattice's partial order). However, it is not yet clear what it means for other operations such as negation or implication to be interpreted within this structure.

Heyting Implication and Pseudocomplements

Implication is the key operation that makes a lattice into a Heyting algebra and enables modeling of first order logic. Without it, we are very limited in the sentences that we can describe with our formal semantics; in fact, we are so limited that we cannot model negation. *Within the lattice structure, the object corresponding to an "implication" relationship between two other objects is a witness to the fact that if one is constructible, the other must also be constructible.*

We want our system to be sound, meaning that we cannot prove false statements.[6] (If we could, then the logic would not be very useful.) In classical logic we can define the implication $a \to b$ as being true if and only if $\neg a \vee b$ is true. This definition makes the most common inference rule, *modus ponens* (i.e. $P \wedge (P \to Q) \implies Q$), sound but it isn't intuitionistically valid as it relies on the use of the law of the excluded middle to prove $a \to b \implies \neg a \vee b$ [32].

Instead, we will depend on a weaker definition of implication defined in terms of the lattice.

Definition 9 (Relative Pseudocomplement)**.** *The witness to a statement $x \to y$ is the greatest*

[6]A corresponding property *completeness* is, in general, unattainable for a logic that is this expressive; c.f. [38].

object satisfying the relation $z \wedge x \leqslant y$ *[32]. This object is known as the "relative pseudocomplement of* x *with respect to* y*."*

From our set example it should be clear that when $a \leqslant b$ then a should imply b. Note that using this intuition the definition preserves *modus ponens*: if $x \to y = z$ then $x \wedge z \leqslant y$. We will also want to have certain other standard properties, such as self-implication and standard distributive properties. Our definition of relative pseudocomplements gives us all of these properties, as shown by Heyting [39]:

1. *Self-implication:* $\forall a,\ (a \to a) = \top$.
2. *Modus ponens:* $\forall ab,\ (a \wedge (a \to b)) = (a \wedge b)$.
3. *Elimination of meets:* $\forall ab,\ (b \wedge (a \to b)) = b$.
4. *Distribution of implication over meets:* $\forall abc,\ (a \to (b \wedge c)) = (a \to b \wedge a \to c)$.

Negation

Let us briefly define negation before returning to the general idea of pseudocomplements. Negation is intuitively understood to mean that the object being described does not exist. This is hard to encode directly, so instead we will use implication and the soundness of our system to write it indirectly.

In any logical system a major consequence of being able to prove false is known as the *principle of explosion*: if one is able to derive a proof of falsity, then any statement can be proved from it regardless of whether it is true or false. We can leverage the principle of explosion to derive an encoding of the negation of an object. We will define the negation of an object as a proof that if that object exists, then we can create an object which has any property we want (i.e. a witness of any proof). This is the standard representation of negation for a Heyting algebra:

$$\forall x, \neg x = (x \to \bot)$$

Note that with this definition one can easily show that $p \wedge \neg p = \bot$, though in general it is not possible to show $p \vee \neg p = \top$—the *law of the excluded middle*—without explicit knowledge about the structure of the particular lattice under study. Our example lattice

$x \to y$ y / x	∅	{A}	{B}	{C}	{A, B}	{A, C}	{B, C}	{A, B, C}
∅	{A, B, C}	{A, B, C}	{A, B, C}	{A, B, C}	{A, B, C}	{A, B, C}	{A, B, C}	{A, B, C}
{A}	{B, C}	{A, B, C}	{B, C}	{B, C}	{A, B, C}	{A, B, C}	{B, C}	{A, B, C}
{B}	{A, C}	{A, C}	{A, B, C}	{A, C}	{A, B, C}	{A, C}	{A, B, C}	{A, B, C}
{C}	{A, B}	{A, B}	{A, B}	{A, B, C}	{A, B}	{A, B, C}	{A, B, C}	{A, B, C}
{A, B}	{C}	{A, C}	{B, C}	{C}	{A, B, C}	{A, C}	{B, C}	{A, B, C}
{A, C}	{B}	{A, B}	{B}	{B, C}	{A, B}	{A, B, C}	{B, C}	{A, B, C}
{B, C}	{A}	{A}	{A, B}	{A, C}	{A, B}	{A, C}	{A, B, C}	{A, B, C}
{A, B, C}	∅	{A}	{B}	{C}	{A, B}	{A, C}	{B, C}	{A, B, C}

Table 5.1: The relative pseudocomplements $(x \to y)$ *for all pairs of* x *and* y *in the* $\{A, B, C\}$ *lattice.*

does exhibit this property, as does the familiar Boolean lattice, with 1 and 0 as the top and bottom elements, respectively, and joins and meets defined in the normal way. The simplest example that does not is a ternary logic consisting of 0, 1/2, and 1. It should not be assumed that the LEM holds for a given Heyting algebra; rather, it must be proved to hold using the existing axioms.

Examples of Pseudocomplements

Returning to pseudocomplements in our example lattice, there are some that are easily identified and some that are more difficult. Among the easier ones are that, for any a and b where $a \leqslant b$, $a \to b$ is simply $\top$. However, if we look at two elements that do not have this relation—say, $\{A, B\}$ and $\{B, C\}$—the pseudocomplement is not readily apparent. Fortunately, we can lean on modern proof tools to help us identify the correct mapping automatically,[7] as shown in table 5.1.

We will examine a few examples to help illustrate the idea of relative pseudocomplements. Let's begin with an easy one. Consider the statement $\emptyset \to \{A, B\} = ?$. To find the answer, we need to construct the greatest object such that the result of meeting it with $\emptyset$ is less than $\{A, B\}$. We know that every object's meet with $\emptyset$ is also $\emptyset$, so the element we want is the greatest element of the lattice, $\{A, B, C\}$.[8]

As another example, consider $\{A\} \to \{B\} = \emptyset$.

1. Goal: find the greatest z such that $\{A\} \wedge z \leqslant \{B\}$.

2. $\forall x, x \leqslant \{B\} \implies (x = \{B\}) \vee (x = \emptyset)$, by examining the structure of the lattice.

[7]To identify the pseudocomplements for this example I used Z3, a Satisfiability Modulo Theories (SMT) solver developed by Microsoft Corporation. The full source of the SMT problem is given in Appendix A.

[8]Note that this is also true for every other element in the lattice, so we can make the stronger statement that $\forall a, \bot \to a = \top$, which is exactly the *principle of explosion* described earlier.

3. The element $\{B, C\}$ has the correct meet: $\{A\} \wedge \{B, C\} = \emptyset$. The elements $\{B\}$, $\{C\}$, and $\emptyset$ do as well but all of these are smaller.

4. The only object greater than $\{B, C\}$ in the lattice is $\{A, B, C\}$. $\{A, B, C\} \wedge \{A\} = \{A\} \not\leq \{B\}$, therefore $\{B, C\}$ is the greatest object whose meet with $\{A\}$ is less than $\{B\}$ and is the desired object.

After this discussion the interpretation of Heyting algebra operations in a set-valued lattice structure should be clear. While defining a logic's semantics in terms of this family of structures is the most natural approach and most applicable to this thesis, they are not the only types of structures that can help to define a logic. Here we use first-order intuitionistic logic, defined through Heyting algebras, because it is the simplest of these which is still powerful enough to use for writing interesting specifications. Many other structures exist that support logics with varying properties, including linear logics that can model the consumption and production of finite resources, modal logics to model actions over time, and the simply typed lambda calculus which provides a natural and powerful way to model computation. Each of these lends more expressive power and captures a greater number of possible behaviors, but at the cost of additional complexity when defining the semantics in a constructive fashion.

For a deeper treatment of this topic I refer the reader to Johnstone [37] who provides perhaps the most detailed description of categorical logic currently available, and Baez [34] who discusses its broad-ranging connections with physics, topology, and computer science.

In the following sections, I will go on to describe the formal properties of the Qualitative State Plan, which shares many properties with the basic sets described here.

5.3 Case Study: Qualitative State Plans (QSPs)

The Qualitative State Plan (QSP) is the fundamental plan representation within Enterprise and thus the focus of attention for inter-component communication specifications. In this section I describe the fundamentals of QSPs, building upwards from the Temporal Constraint Network to include constraints on the state parameters of the controlled system. In the process I explore the algebraic properties that are of interest in demonstrating that

QSPs form a Heyting algebra, which is presented in the following section.

5.3.1 Temporal Constraints

Scheduling is a fundamental part of almost every real-world planning task. In general, our problems require that the system produce a schedule of actions that meet a set of user-provided timing constraints. These constraints enforce conditions such as "after shutting down the computer, wait 10 seconds for the process to complete" or "within a day after beginning execution, you must reach the objective location." A QSP embeds a description of the schedule requirements using a *temporal constraint network (TCN)* [31], consisting of a set of *events* representing independent points along a real-valued timeline augmented with a set of temporal constraints between pairs of events. Each of these constraints contains an ordered pair of start and end events (E_s, E_e) along with a corresponding set of time intervals restricting the duration between the start and the end events expressed as linear inequalities such as $-\infty \leqslant E_e - E_s \leqslant \infty$. A temporal constraint whose interval set is a singleton is considered "simple," and a network consisting of only simple constraints is a *simple temporal network*.

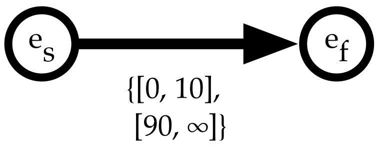

Figure 5-2: A basic temporal constraint network consisting of two events and one constraint.

These sets of intervals form the heart of the constraint network and are the basis for its algebraic properties. An individual constraint is considered "satisfied" by a timing assignment to its events if that assignment makes the duration between the start and end events fall inside one of the intervals in the constraint's interval set. To help illustrate the idea, consider the simple constraint network shown in figure 5-2. We have two events, the start event e_s and finish event e_f, joined by a constraint restricting the timing difference, $\Delta_{s,f} = e_f - e_s$, between them to $10 \leqslant \Delta_{s,f} \leqslant 30$ time units or to $90 \leqslant \Delta_{s,f} \leqslant \infty$ time units. This means that, whatever value we give e_s, it must be at least 10 units before e_f and e_f must follow either no more than 30 or no less than 40 units after. In this case, the assignment $e_s \mapsto 5$, $e_f \mapsto 25$ would satisfy the constraint while $e_s \mapsto -10$, $e_f \mapsto -50$ would not.[9]

Writing the set of constraints between events as $C(e_s, e_f)$, the constraint network as a whole is satisfied if, for every pair of events with a constraint, the time between the start and end events satisfies any of the the intervals of the constraint between those two events. This is reflected in the following formula:

$$\bigwedge_{e_s, e_f \in E} \bigvee_{c \in C(e_s, e_f)} l_c \leqslant e_s - e_f \leqslant u_c$$

In their treatment of these structures Dechter, et al defined operations corresponding to joins and meets for temporal constraints as follows [31]:

- The join of two constraints accepts only values that are accepted by one or both of the two source constraints.
- The meet of two constraints accepts only values that are accepted by both of the two source constraints.

In addition, Dechter, et al define a partial ordering relation among temporal constraint networks: between a pair of constraint sets A and B sharing start and end events, A is equally or less permissive than B if every assignment accepted by A is also accepted by B [31]. In general this will require that the events contained in B are a subset of those in A, and that every decision already made in B about the assignments to the temporal variables is preserved in A. When this condition holds we can say that $A \leqslant B$ or that A "refines" B. Formalizing this definition we say that for two constraint networks C and D, C refines D if and only if C contains every event in D and for every assignment to the temporal variables that C accepts, the same assignment is also accepted by D. Two constraint networks are *equivalent* ($C \equiv D$) if and only if they each refine the other.

These operations, combined with the refinement ordering relation, induce the lattice structure required for a Heyting algebra. In fact, if one considers an alternative representation of the constraints—that is, that the constraints are given by the explicit sets of

[9]Every constraint also has a dual, where the start and end events are swapped and every interval in the constraint is reflected across the origin, so that $[a, b]$ becomes $[-b, -a]$. These duals can often be used to make the network well-formed for the purposes of computation, and in particular are useful for ensuring that a temporal constraint network's directed graph is acyclic.

values they accept—the lattice is clearly the same as the set-valued structures examined previously, with intersection and union given their normal set-theoretic definitions, and bounded at the bottom by the "empty" constraint set accepting no assignments and at the top by the "universal" constraint set accepting any assignment.

Viewing sets of linear constraints in terms of the sets of accepted assignment values to each variable is a natural and valuable perspective making the visualization and intuition of these structures more apparent. We will lean on this viewpoint in the next section to define state constraints and the lattice of QSPs.

5.3.2 State Constraints

Whereas temporal constraints limit the potential assignments of time to events in the problem, a state constraint will restrict the values that a variable representing the physical properties of the system can take.[10] At their most basic, state constraints, like temporal constraints, are defined using linear inequalities and define convex polytopic regions within the state space (which may or may not be bounded). For example, a state constraint may define the feasible region of acceleration for a vehicle, restricting it to the region between $\pm 7\text{m/s}^2$. Others may define the area of operations for the system by specifying that the position state variables remain within a given half-space (see figure 5-3). Combining groups of these constraints is simply a matter of taking the intersection of the polytopes they define, making it easy to create composite constraints acting over multiple variables or bounding the same variables from multiple directions.

While very convenient computationally and easy to reason about, if we restrict ourselves solely to conjunctive sets of linear inequalities we will not be able to model obstacles; so, we must extend the formalism to include disjunction. This reimagines the model as a *set* of polytopes containing feasible regions rather than a single polytope, and membership in any one of these regions is acceptable. The true system state may pass through a variety of these polytopes over time, in order to perform mode switching and to avoid obstacles, passing between the feasible regions where they overlap.

Without some connection with time state constraints are unsuitable for a goal speci-

[10]In the planning community the general practice is to define state variables as distinct from control variables, with the assumption that the former are not directly assignable by the planner. I take the view that because the state and control variables constrain each other exactly, they are effectively the same class of variables and can be treated the same for the purposes of mathematics.

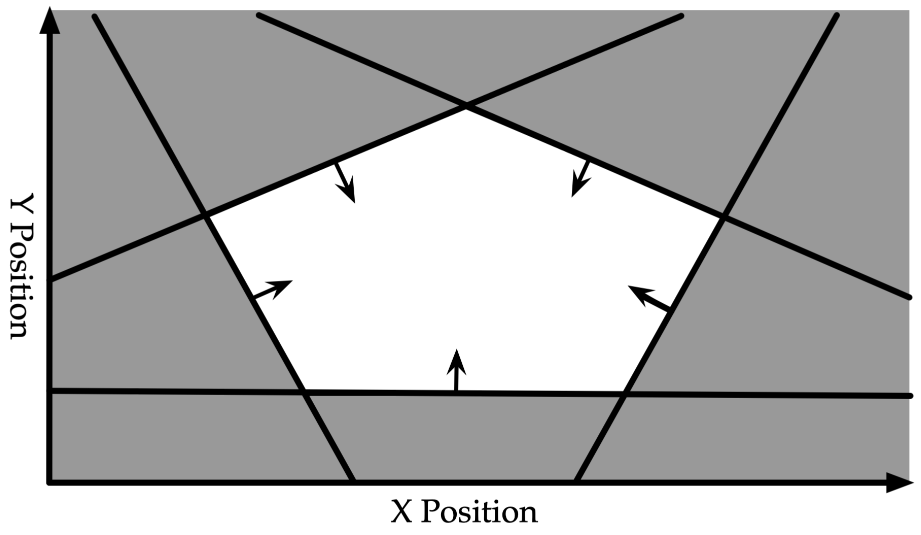

Figure 5-3: A region defined by the intersection of half planes.

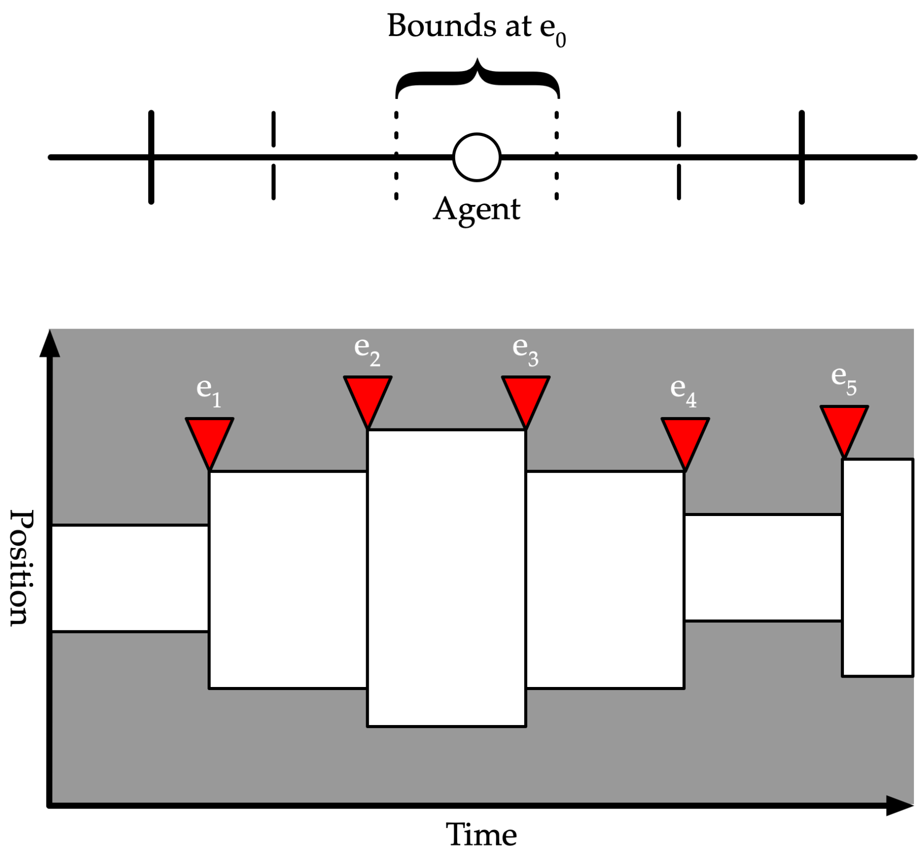

Figure 5-4: In this illustration, an agent must remain between walls which move over time.

fication or for describing changing environments or vehicle dynamics. To avoid this, we will embed the state constraints within a temporal network by denoting specific start and end events between which a given state constraint is considered "active" and must be obeyed. The period between these events, combined with the corresponding state constraint, forms an *episode* in the plan. Consider the example given in figure 5-4, where an agent in a 1-dimensional world must remain between two walls, which shift position over time.

At e_1 and e_2 the walls move outward from their starting position, at e_3 they return, and so on; this is represented with state space constraints restricting the agent to be between the walls as they are positioned at these various times. Each of these combinations of a state constraint and temporal interval forms an episode. Note that the feasible region for the entire plan is the result of combining the feasible regions for each episode. The system state trajectory must pass between the regions where they touch at event boundaries, here represented as vertical lines. These interfaces between episodes correspond to the "meet" of the episodes at that point. The system may otherwise occupy any space within the feasible region at a given time point.

A similar set of regions can be constructed for higher dimensional systems by "gluing" viable state polytopes together at event boundaries, themselves represented as hyperplanes which are normal to the time axis. This *geometric interpretation of state constraints* gives rise to the idea of a "flow tube" introduced by Li [40], drawing upon previous work by Zhao [41] and Hofmann [42]: an n-dimensional polytope that compactly encodes the set of valid trajectories that the system state may take between two timepoints. Flow tubes provide an intuitive representation for episodes, hinting at opportunities for abstraction and concretization. We will discuss these opportunities in more detail in Section 5.4.

5.3.3 Definition of the Qualitative State Plan

Combining temporal and state constraints as described above yields a composite structure suitable for encoding both models of systems and goals: the Qualitative State Plan (QSP). As defined by Léauté [8], QSPs contain:

- A set of *events* that can be assigned specific points along a real-valued timeline.
- A set of *temporal constraints*, sets of linear inequalities providing bounds on the durations between the events.

- A set of *episodes*, each a set of state constraints between a specific pair of events. In a QSP these constraints may additionally be quantified with one of a "for all" (∀), "there exists" (∃), "beginning at", or "ending at" quantifier that state, respectively, that the system must respect the state constraints during the entire duration between the start and end events, that it must satisfy them at least once during that period, that it must satisfy them at the instant of the start event, or that it must satisfy them at the instant of the end event [8].

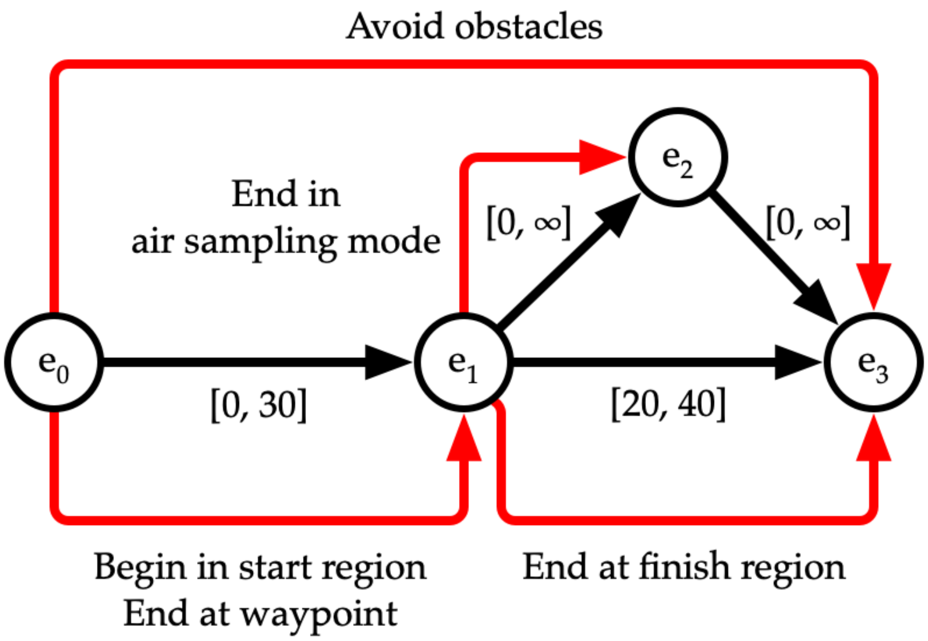

Figure 5-5: An example QSP describing a high-level plan for a fictitious UAV.

It can be convenient to represent QSPs in a graphical form, as shown in figure 5-5. Graphically, these elements form a directed graph with labeled edges: the events form the vertices while the temporal constraints and episodes are edges between their start and end events labeled with their constraint sets. The edges representing temporal constraints are directed, such that the decision variable representing the time difference between the two vertices is defined as $e_f - e_s$, with e_s being the origin vertex and e_f the destination vertex, as in a Temporal Constraint Network.

To make the structure easier to reason about we additionally impose that this graph to be acyclic, and have a single start and end; thus we will define a *well-formed QSP* as one in which:

- There exist distinct start and end events for the QSP, such that the start is followed by or at the same time as, and the end preceded by or at the same time as, all other events in the QSP.
- There exists a path in the constraint graph from the start event to every other event in the QSP.
- The end event is reachable within the constraint graph from every other event in the QSP.
- No (directed) cycles exist within the temporal graph. Some QSPs which would otherwise contain cycles can be made acyclic by reversing some of the constraints, using the duality property of temporal constraints discussed earlier.

5.3.4 Use of Constraint Systems for Models, Goals, and Solutions

Generally, model-based planning systems will make a distinction between several different models that are used as input to the planner. These include the *plant model*, the *environment model*, and the *goal specification*; of these, only the last has generally been given as a time dependent constraint problem, the others being represented either as time-invariant equations or in a geospatial encoding such as GeoJSON. However, we will see that all three of these models can be expressed using the tools a QSP makes available, as can the solution to a planning problem. This same approach can be generalized to other structures supporting similar features as well.

The first of our models, the *plant model*, defines the relationships between various state variables to encode the dynamics of the system under control. Plant models are generally taken to apply across the entirety of the planned duration, as they represent intrinsic properties of the system. A QSP can model this style of constraint as a $\forall$-quantified episode between the start and end events. The *environment model* encodes regions that the system must either avoid or remain in; obstacles may be encoded in their dual form, as a choice between half-spaces that the system may be in.[11] QSPs are able to encode this type

[11]Environments are not generally stored as QSPs, but rather given a geometric representation that is more compact and easier to edit with available tools. However, there is a useful isomorphism between convex polytopic environments and disjunctive linear constraint systems that allows encoding them as QSPs; we will ignore the alternative representation for convenience.

of constraint similarly with a ∀-quantified episode, relying on the disjunctive form of the state constraint to provide the various choices.

The final model encodes the *goal specification* for the system as a set of constraints on the high-level system behavior. These generally provide the bulk of the temporal constraints and the initial set of state constraints defining the objectives of the system user. Goal specifications will constrain the starting and ending conditions of the plan at a minimum, and generally provide intermediate events with corresponding state constraints defining the high-level properties of the desired system state trajectory.

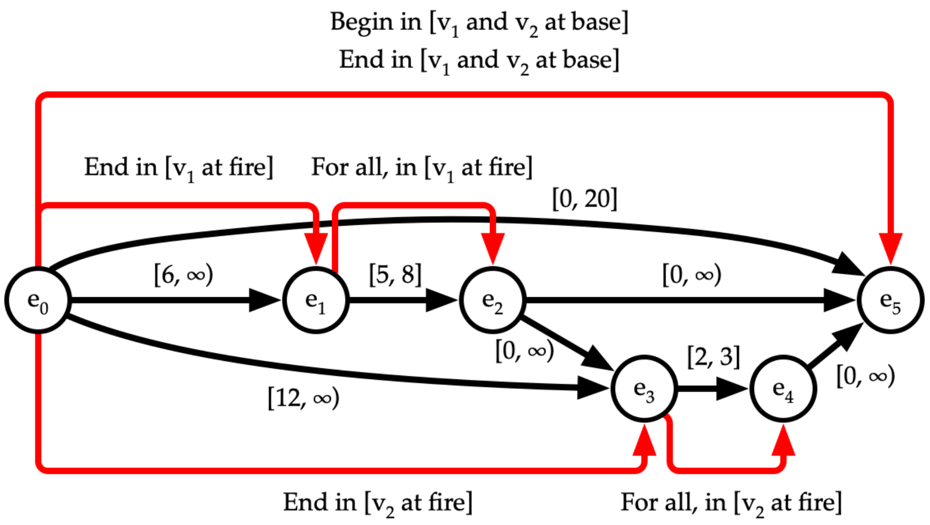

Figure 5-6: The goal model QSP for the firefighting example.

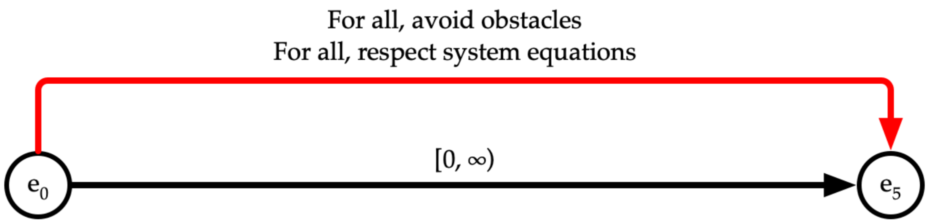

Figure 5-7: The plant and obstacle model QSP for the firefighting example.

As an illustrative example, I will borrow a firefighting scenario described by Léauté [8] in which a pair of autonomous vehicles must depart from their respective home bases, enter the vicinity of a forest fire, drop water on it, photograph the damage, and then return to their bases within a 20 minute time window. The goal QSP (depicted in figure 5-6) encodes these constraints, requiring the vehicles to be at certain places at certain times, and enforcing a timing relationship between their activities so that the fire is extinguished before the photography begins. This object does not include constraints on the system dynamics or obstacles to be avoided, so another QSP must be provided that specifies these (shown in figure 5-7).

A solution to this planning problem is an assignment of time to the events along with an assignment to the various state variables over time.[12] Assignments can either be so tight that they only admit a single value, in which case the plan is "grounded," or loose enough to give an executive the ability to modify the plan in response to state updates during the execution phase. If an executive is available, the ideal is a "least-commitment" plan: one that is constrained enough that the admitted solutions will all achieve the goal, but with the maximum amount of flexibility allowed otherwise. We will revisit this idea in the context of the Heyting algebra in the next section; for the time being, it is sufficient to state that these results are both representable as QSPs by adding events and constraints that further refine the goal and model specifications to include only the desired solutions.

QSPs have been applied to both trajectory optimization and activity planning problems, with examples such as Kongming [40], Scotty [9], Sulu [8], and tBurton [43]. The details of these planners and their approaches to QSP planning are beyond the scope of this thesis; let it suffice to say that the constraint system models used here are rich and applicable to a wide variety of planning domains.

5.3.5 The Heyting Algebra of Qualitative State Plans

The previous sections have discussed how Heyting algebras can be used to give a semantics for first-order intuitionistic logic and defined the Qualitative State Plan. We will now proceed to combine these two ideas and define a logic which allows us to write concise specifications of planner behavior by connecting logical operations to common concepts from the planning community, such as relaxations and conflicts. In a nutshell, these connections are:

1. Implication encodes when the solutions to one QSP are also solutions to another.

2. Joins encode constraint relaxation.

[12]For the purposes of this thesis, I am ignoring the distinction between state and control variables. This is a reasonable thing to do, because we are concerned here with what a feasible solution looks like in terms of the constraint system properties, and not with the details of how to generate it. Because a QSP which is inconsistent between the controllable and uncontrollable variables is equivalent to $\perp$, any valid QSP must also have valid control variable assignments unless the control equations are under-constraining.

3. Meets encode the addition of new constraints.

4. Negation encodes conflict kernels.

Moving on to the definition of the Heyting algebra for QSPs, we will first give a high-level outline of the approach before proceeding to define the necessary operations in more detail. Qualitative State Plans, as implied by their geometric interpretation as flow tubes, describe sets of possible assignments to the state and temporal variables of the system under control. Fundamentally a QSP is simply a compact encoding of this set. To define the Heyting algebra over the set of QSPs, then, we will take a similar approach to the case of temporal constraint networks and view the QSPs in terms of their respective solution set.

In this view, a QSP A is less than or equal to another, B, if any assignment of time or state variables accepted by A is also accepted by B. Previously in the case of temporal networks we called this relationship a *refinement*; the term applies here as well, reflecting the fact that A is restricting the possible solution set $\mathrm{sol}(B)$ by making commitments to certain assignments that B left undecided.

Definition 10 (Ordering Relation of QSPs)**.** *For two QSPs,* $A \leqslant B \iff \mathrm{sol}(A) \subseteq \mathrm{sol}(B)$.

Definition 11 (Equivalence of QSPs)**.** *Two QSPs are equivalent if and only if they are mutually refining:* $A \equiv B \iff (A \leqslant B) \wedge (B \leqslant A)$.

Definition 12 (Equivalence Class)**.** *An equivalence class is a set of objects which are all equivalent to each other.*

Mutual refinement of two QSPs does not necessarily imply that they have the exact same structure. They must share the names of their events—up to a form of α equivalence—but they may have more or fewer temporal and state constraints as long as the accepted variable assignments are the same. This distinction is not particularly interesting from a theoretical standpoint, so we will define two QSPs as being equivalent if and only if they are mutually refining, and define the QSP lattice in terms of equivalence classes of QSPs rather than specific objects themselves.

We have been defining properties of QSPs in terms of their solution sets, but in doing so there is a danger that we accidentally make claims about QSPs which are true for sets of state trajectories but that are not true for the constraint representation. We can avoid this by establishing that there is a transformation between QSPs and their solution sets, such that every QSP has a solution (which may be empty) and every solution has a QSP.

Given a solution set—i.e. a set of assignments—for an equivalence class of QSPs, it is possible to re-encode it as a QSP by creating a set of events mirroring that of the solution set, adding a temporal constraint for each possible timing assignment between each pair of events, and then adding state constraints in a similar manner. This approach does not yield a particularly elegant or minimal encoding; the result can be improved by reducing the temporal constraints to a set of disjoint intervals and similarly with the state constraints.

Now that we have established that working with the solution sets is safe, we can define a Heyting algebra for QSPs easily. We will consider the equivalence classes of QSP solutions as the objects of the lattice, ordered by subset inclusion (equivalent to QSP refinement). Boundedness is given in the same way that we described for temporal networks: at the bottom is the empty solution set, whose equivalence class contains all QSPs admitting no solutions, and at the top the universal set that accepts any assignment to the state and temporal decision variables.

In the solution space, meets and joins are defined in the standard way for sets, with intersection and union, respectively. These operations have interesting interpretations in terms of the QSPs they operate on: a meet corresponds to the merging of two QSPs such that the result contains all of the constraints of its parents, while a join is the relaxation of the constraints—both implicit and explicit—that are not shared between two QSPs.[13] Relaxation of constraints is a straightforward operation,[14] generally involving either deleting a constraint edge or adjusting its inequality sets. Merging of two QSPs is slightly more involved, requiring that shared events are combined along with the corresponding state and temporal constraint sets.

Relative pseudocomplements are defined in a similar fashion, and allow us to model the implication between two QSPs when one refines the other: $\forall ab, a \leqslant b \implies a \to b$. Another interesting interpretation of implication arises when we consider the definition of negation, and gives us a way to model conflicting constraints. Recall the definition

[13]It is easy to create two QSPs that share only a small portion of their explicit constraints but have the same solution set. Considering the implied constraints as well, the two are the same as captured by the equivalence relation defined in this section.

[14]Mathematically speaking, that is; finding the right relaxation is not easy in practice.

$\neg x = x \to \bot$. This means that the object $\neg x$ is a proof of the fact that every time we have the conjunction $x \wedge \neg x$ we will always derive $\bot$ from it. In this sense, $\neg x$ contains conflicting constraints which can't be combined with the constraints in x while still having solutions. In addition, recall that Heyting pseudocomplements are the greatest object in the lattice with this property, so $\neg x$ is the most permissive set of QSPs which are incompatible with x; i.e. they are minimal conflict kernels.

In summary, the Heyting algebra gives a formal way to connect concepts familiar to the planning community to operations in first-order logic so that they can be modeled concisely and used to write specifications of planner behaviors. Recall the list of concepts from the beginning of this section:

1. Implication encodes when the solutions to one QSP are also solutions to another.
2. Joins encode constraint relaxation.
3. Meets encode the addition of new constraints.
4. Negation encodes conflict kernels.

In the next section we will examine how to evaluate first order logic formulae within this structure, and what types of objects are produced by doing so.

5.3.6 Evaluating FOL Formulae in the Heyting Poset

Recall from earlier in this chapter that the signature of a first-order logic contains a set of atomic objects, a set of logical connectives, a set of variables, a set of relation symbols, a set of function symbols, and a pair of quantifiers. To interpret the logic in terms of a Heyting algebra, we need to give a definition of how these pieces map to objects in the lattice.

In the absence of quantifiers—i.e. when there are no free variables—reducing a propositional logic formula to a truth value is typically done recursively, by reducing each clause until atomic values are reached and then proceeding using the inference rules of the logic to join the atoms together. A similar approach works for interpreting formulae in our Heyting lattices. Atomic propositions refer to specific objects in the lattice, and the connectives $\wedge$, $\vee$, $\to$, and $\neg$ can be easily interpreted by locating the corresponding value in the lattice according to the operator's definition discussed previously. Function and relation symbols are also simple to evaluate when there are no free variables: evaluating a function returns a specific object within the lattice, while a relation is simply a function of two inputs whose output is restricted to either $\top$ or $\bot$. When fully reduced, the resulting object is a witness to the truth (or falsehood) of a given formula.[15]

Quantifiers introduce free variables that must be grounded in order to construct witness objects. For existential quantifiers, a proof of $\exists x, \phi(x)$ is given by a pair $\langle x', \psi \rangle$ where x' is an object in the lattice and ψ is a witness to the proof that $\phi(x')$ is true [45]. This essentially entails finding some particular value that makes the formula true. Universal quantifiers are more complicated: they require defining a function assigning to each value in the quantified set a proof object for the formula being quantified over when the quantified variable takes that value.

5.4 Extensions: Abstraction and Refinement of Constraint Systems

In this chapter we have examined a specific family of mathematical structures (Heyting algebras) and used them as a tool to define the semantics of a particular logic, intuitionistic first-order logic, in terms of the Qualitative State Plan. As discussed above, QSPs are contained within the family of linear constraint systems operating over continuous time and space. Of course, these are not the only constraint systems that can be constructed, nor are they appropriate for all circumstances. The design space for these structures extends both higher and lower in abstraction, and similar techniques for deriving the logics they support also apply in those domains. I will not attempt to provide a comprehensive treatment of this topic here, but rather I will show some examples of more abstract and more concrete structures and their relationship to the QSP, with referrals to the reader as to where they can investigate further.

To begin we will consider the case of making a constraint system more concrete by discretizing it. This is a common technique performed in trajectory optimization; in particular, I will refer to the type of time discretization used by the Sulu planner described

[15]Note that as a first-order logic including both function and binary relation symbols, the decidability of arbitrary formulae is not guaranteed, as established by Trakhtenbrot [44].

by Léauté in [8]. As part of its problem formulation, Sulu divides its planning horizon into a uniform series of time intervals, between which the control variables to the system are held constant. This discretization enables the QSP planning problem to be formulated as a Mixed Integer Linear Program (MILP) and solved using standard techniques. Any solution to this discretized problem formulation is also a solution to variants where control variables may be modified at any time (or in the limit, continuously) to achieve the desired trajectory. This is because discretized problems form a proper subset of the continuous problems, with additional constraints added to enforce that the events occur on certain time boundaries.

We can define operations to convert continuous-time problems into discrete-time ones, and vice versa, by adding or removing constraints. By convention these are referred to as α (the *abstraction* function) and γ (the *concretization* function). Together these form a *Galois connection* between the abstract poset and the concrete poset if the following relation holds [46]:

$$\forall ac, a \leqslant \alpha(c) \iff \gamma(a) \leqslant c$$

Galois connections enable the system designer to choose their level of abstraction for the problem to fit the technology available for solving it, while also enabling flexibility in specification. Concrete problems are generally easier to formulate than abstract ones, but abstract problems can be easier to describe. For example, if the discretization is auxiliary to the fundamental problem being solved, then specifications can be made in the "abstract" continuous space to describe the concrete implementations, giving more flexibility to those implementations and easing the specification burden.[16]

Unsurprisingly, there are more abstract structures supporting Galois connections to QSPs. A promising one is the category *nCob* of generalized $(n-1)$-dimensional manifolds and their (n)-dimensional cobordisms. This category contains $(n-1)$-dimensional manifolds, the disjoint unions of which form the boundaries of manifolds one dimension higher [34]. Cobordisms are combined at their boundaries to create "tubes" through the (n)-dimensional space.

The analogy to the geometric interpretation of a QSP should be fairly clear: we have

[16]As an interesting example of this approach, c.f. Smith's work "Toward the Synthesis of Constraint Solvers" [46], in which Galois connections are used to derive provably optimal constraint solvers.

relaxed the requirement that our constraints be linear, and created a more abstract notion of flow tubes joined at certain planar boundaries. QSPs are the subset of these manifolds that include a linearity criteria, forcing the manifolds to be prismatic. Relaxing that constraint allows modeling of more complex vehicle dynamics, obstacle regions, and goals.

Just as it supports more expressive objects, *nCob* also supports a more expressive internal logic. As a *closed symmetric monoidal category*, its internal logic is *multiplicative intuitionistic linear logic* (MILL). Linear logics are strictly more powerful than first-order logic—one can encode FOL in LL, but not the reverse [47]—and are able to model the transformation of finite sets of resources. To illustrate the idea, consider the intuitionistic model of eating a cake: let the statement $\mathtt{cake} \implies \mathtt{happy}$ be true if if eating a cake can make you happy. In traditional intuitionistic logic, including the variants defined above, one can infer from $\mathtt{cake}$ and $\mathtt{cake} \implies \mathtt{happy}$ that $\mathtt{cake} \wedge \mathtt{happy}$—"you can eat your cake and have it too." In a linear logic this type of resource can be modeled, such that in the process of deducing $\mathtt{happy}$ the input $\mathtt{cake}$ is consumed and no longer available.

Future work may explore the connections between linear logic and plan representations in more detail. I anticipate that this would be a fruitful area for deeper study, enabling more powerful specifications of system behavior.

Chapter 6

Application to Modeling and an Enterprise Case Study

In the last two chapters we developed a semantics for first-order logic defined in terms of the constraint problems used to represent models, goals, and solutions within a planning system, and described a particular system, *Enterprise*, that uses such a structure, the Qualitative State Plan (QSP), to communicate between its component planners. We will now connect these pieces via a case study. This chapter will demonstrate how a model of *Enterprise* is constructed to match the formalism introduced in the first chapter, how component specifications are given for key properties, and discuss what remains in order to build a proof that the overall system structure meets these requirements.

6.1 Abstract Architecture of Enterprise

Before diving in to building a model of *Enterprise*, I will first establish the appropriateness of the formalism described in Chapter 1 as a modeling approach. Recall the key conditions: first, that planners communicate using a single message type, T; second, that they act as functions of type $\mathsf{T} \times \mathsf{T} \to \mathsf{T}$, transforming these messages while retaining their type; and third, that they are able to be wired together into a tree structure by substituting implementations of these functions for symbolic "holes" in the bodies of other functions. All of these criteria are met by Enterprise. Our message type will be the QSP, which planners both consume and produce. Due to the mechanisms for managing state and ensuring solution invariance with regards to it, the planners can safely be considered as though they were pure functions of their inputs. We are able to combine planners by giving them the network address of each subplanner so that the parent may call the APIs of the child.

Enterprise is flexible enough to take on a number of different forms. The first is a simple design, where only a single planner is required. For this architecture, we will have a single abstract component. The instantiation table is similarly simple, with a single mapping of that abstract component to a concrete component that does not contain any holes.

The result of instantiating this architecture is, as one might guess, a single standalone planner—hardly the most interesting design, but we need to start somewhere. A more sophisticated design may start with the same abstract architecture, but with an instantiation table mapping the abstract component to a concrete one that requires additional pieces; in this way, we can describe a multi-level planning system that places an activity planner (in this case, a variant of Kirk [28]) in a position to coordinate the creation of sub-plans that are dispatched to a path planner.

This type of design is as unconstrained as possible: the only specification given is on the top-level behavior as exhibited by the single abstract component. While this is conceptually quite simple, the lack of constraint in the system implies a lack of support for reasoning about appropriate decompositions; that is to say, we may not be able to easily decide what planners are needed to satisfy the overall specification. This design knowledge is captured in the choice of mappings in the instantiation table rather than made explicit in the architecture. Contrast this approach with the less flexible but more explicit design that calls for subplanners with a specific set of properties. A more constrained approach gives clear guidance about where the inter-component interfaces are and what makes a planner acceptable for use in each of those contexts.

6.2 Modeling Activity and Path Planners

Generally speaking there are two major categories of planners in *Enterprise*. Activity planners decide sequences of high-level actions that are necessary to progress from an initial state to a goal state, using an action model describing what the system under control can and cannot do. The result is a schedule of tasks that the system must perform, which may be flexible or may have explicit timing assignments. A path planner, in contrast, generally operates at a lower level using a more detailed model of the agent's physical properties, with the goal of creating a plan of control inputs over time that direct the state trajectory of the system to evolve towards the desired final state. Said another way, activity planners operate by scheduling abstracted tasks while path planners schedule control inputs. The distinction is subtle but important, as each type of planner will have greater expressive power at a different level of abstraction, and different planner types will be suited for different roles within a composite architecture.

6.2.1 Common Properties

There is a basic set of correctness properties that applies equally to activity and path planners. Of these, the most fundamental are *correctness with regards to constraints* and *termination*. The first requires that the planning system—represented in equations as s—not ignore any constraints given in the problem definition, i.e. any solution produced by the planner must satisfy both the system model and the goal specification. (If there are no other solutions, the empty solution may be returned; this is a solution to any goal, but is not a particularly useful one.)

Definition 13 (Constraint Correctness)**.** *A planner is* correct with regards to constraints *if there exists a proof that for any combination of model and goal specifications accepted by the planner, the solution returned by the planner entails both the model and the goal. This property is true if and only if there exists a proof that the planner satisfies the following sentence in the specification logic:*

$$\text{constraint\_correct}(s) = \forall mgp : T,\ (s(m, g) \equiv p) \rightarrow [(p \rightarrow m) \wedge (p \rightarrow g)] \tag{6.1}$$

Definition 14 (Termination)**.** *A planner is* proven to terminate *if there exists a proof that it will return a solution for any combination of model and goal inputs, given sufficient computation time.*

For any planner shown to meet its specification, the termination property is implicit due to the first-order structure of our logic. We have no way of representing that a computation may not terminate, such as returning a function or a continuation; proofs of correctness must establish that a result will be computed given enough time. This is simultaneously a very useful property, and a restricting one: it means that we can always count on our systems to give us either a solution or indicate that one does not exist, but we cannot represent systems that operate continuously over infinite time horizons. Extending the logic beyond batch planning systems, to include continuous monitoring and execution systems, is a worthwhile goal for future research.

More advanced properties can also be represented. Consider a planner used in one of the aforementioned continuous execution systems. In this case, we may want a planner that makes the fewest decisions necessary to ensure that our goals are met, leaving the most flexibility to a run-time executive to adjust the schedule of events to match the situation as it evolves. Such a planner is said to be creating a *least commitment* plan.

Definition 15 (Least-Commitment Plan)**.** *A* least commitment plan *is the least constrained plan that satisfies a given model and goal specification. In terms of the plan lattice, it is the meet of the goal and model objects:*

$$\text{least\_commit}(m, g, p) \triangleq p \equiv (m \wedge g) \tag{6.2}$$

At the opposite extreme, if there is no run-time support available, such as if a robot can only be programmed with predetermined actions on a fixed schedule, the system may need to produce a *grounded plan*.

Definition 16 (Grounded Plan)**.** *A* grounded plan *is the most constrained plan that both satisfies and a given model and goal specification and is not empty. If it exists, in addition to the normal correctness specification it also satisfies the sentence:*

$$\text{grounded}(p) \triangleq \forall mg : T,\ \big[\big(\forall y : T,\ (y \to m \wedge y \to g \wedge \neg(y \equiv \bot)) \to (p \to y)\big) \wedge \neg(p \equiv \bot)\big] \tag{6.3}$$

Intuitively, a grounded plan is one that has had all of the decisions made ahead of time. Adding any more constraints to the problem will either have no effect or lead to a conflict.

Depending on the structure of the planning system's architecture, these properties may be preserved (or not) when components having them are combined. The next section describes two major roles that components play in architectures, and how that affects their specifications.

6.2.2 Leaf and coordinator planners

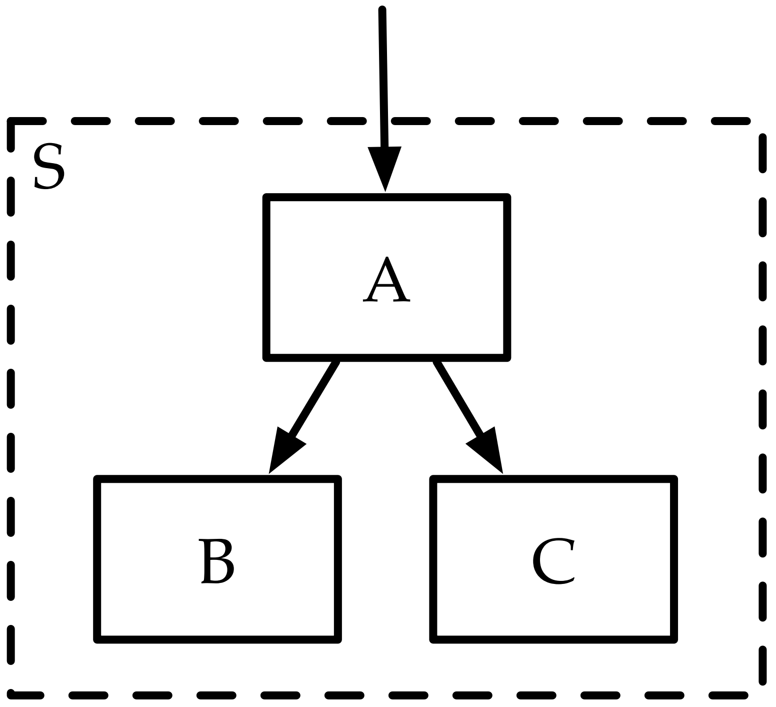

Figure 6-1: An example system containing both coordinator (A) and leaf planners (B, C).

Decomposing the system into a tree implies that planners can be given one of two distinct roles: that of a *coordinating planner* and that of a *leaf planner*. Coordinating planners are responsible for dispatching appropriate sub-problems to their children and then recombining the results in a principled way that builds a solution to their original input. Leaf planners, having no children, are independently responsible for providing solutions to the problems given to them. Figure 6-1 shows a basic system containing both a coordinating planner, A, and a pair of leaf planners, B and C.

To view this idea from another angle, consider the system from the problem decomposition perspective as shown in figure 6-2. There must be a certain relationship between the original input problem and the subproblems dispatched to the subplanners in order for solutions to the latter to be useful. These subproblems are not necessarily going to be full

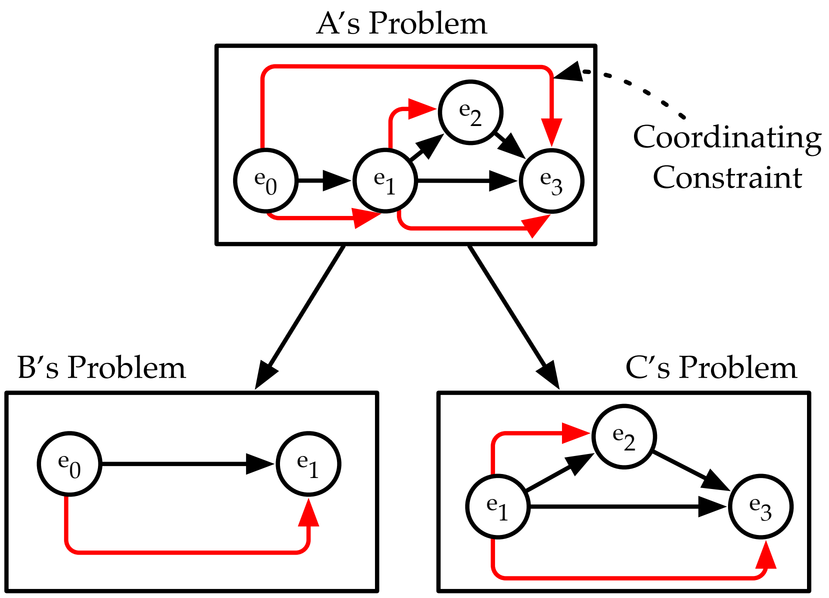

Figure 6-2: A possible problem decomposition for the example architecture.

refinements of the original; for example, a path planner that is used to plan a trajectory between two points may be tasked with a subproblem, the solution to which does not fully solve the original problem. The task of a coordinating planner is to dispatch appropriate subproblems to its child planners and recombine them to produce a complete plan that entails the original input problem—or to decide that no such plan is possible. Leaf planners have a much simpler job, as they simply need to satisfy their own specifications and provide correct outputs to their callers without invoking other planners.

A trivial, and not particularly useful, coordinating planner may do nothing to transform its input and simply pass the problem directly along to a subplanner. A more useful coordinator may modify the QSP by adding additional events and activities, such that the new QSP is a refinement of the old, and dispatch individual activities to subplanners. A correct composition of the results then depends on whether the QSPs can be combined safely with regards to temporal and state constraints. Coordinating planners will often need to manage information about *coordinating constraints*: constraints in the original problem which cross the boundaries between subproblems in the decomposition. This type of scenario is illustrated in figure 6-2, which shows a possible decomposition of a problem that introduces a coordinating constraint not captured in the subproblems. If the decomposition is not done appropriately, the resulting solutions to the subproblems may be impossible to reconcile.

One might reasonably ask, what guidelines exist to ensure that decompositions are safe? We can describe, at a high level, what kinds of operations we need to find and use these specifications as a starting point for either a traditional algorithm design process or a stepwise refinement approach. To a first approximation, we want to ensure that there exist operations *decompose*, with type $T \to T \times T$; *refine* ($T \to T$); and *recompose* ($T \times T \to T$) such that the diagram shown in figure 6-3 commutes.

$$\begin{tikzcd}
A \arrow[rr, dashed, "\text{refine}"] \arrow[d, "\text{decompose}"'] & \geq & A' \\
(A_1, A_2) \arrow[rr, "\text{refine} \times \text{refine}"'] & & (A_1', A_2') \arrow[u, "\text{compose}"']
\end{tikzcd}$$

Figure 6-3: Commutative diagram specification for decompose, refine, and compose.

In essence, we want to be able to break a problem into two (or more, through repeated decomposition) subproblems, solve them individually, and then combine them back to a solution to the overall problem. This is the specification for the standard divide-and-conquer method. A trivial implementation of decompose may opt to always return a pair containing the original problem and the universal problem (which accepts all solutions); this could satisfy the specification when paired with an implementation of compose that just returns the first element of a pair (i.e. the refinement of the original problem). However, this isn't a very useful design as it doesn't reduce the complexity of the problem at all. A better option will reduce the problem complexity so that the subproblems each capture an even portion of the constraints and have similar levels of difficulty.

The particulars of how to implement these operations is dependent on the data structure being used and the ways in which its constraints interact. For the QSP, our example structure, the temporal constraints are simple to decompose because they have well-defined coordination points—the events—and interval arithmetic can be applied to introduce new ones convenient for the decomposition. As an example, consider the following procedure for finding a decomposition of a temporal network:

1. For each event e in the graph, create two events e' and e'' connected by the constraint set $\{[0, 0]\}$. Connect e' as the target for any constraints that ended at e, and e'' for any constraints originating from e'' as represented in the directed graph interpretation of temporal constraint networks.

2. The temporal network is now a directed graph with twice the number of nodes. Weight each edge by the sum of the sizes of the intervals contained in its constraint. Find the most balanced min-cut of the graph, minimizing the weight of the cut edges, such that dropping the selected edges separates it into two evenly-sized subgraphs. The events joined by the dropped edges are *boundary events* that interface between the decomposed graphs.

3. In each of the new graphs, for each pair of events e', e'' connected by the constraint set $\{[0, 0]\}$, merge the events into a single event e receiving the constraints incoming to e' and sourcing the constraints outgoing from e''.

4. Solve the temporal constraint networks independently.

5. Recombine the solutions by reintroducing edges between the boundary events, merging these events as in step 3, and adjusting the timing of the "latter" constraint network to account for the timing assignment to the boundary event made in the "earlier" one.

This definition satisfies the diagram, and works well for temporal constraint networks. However, it does not take into account interference between state constraints and thus cannot be applied to generalized QSPs. State constraints are more challenging than temporal constraints in large part because they apply continuously over periods time, rather than at discrete instants[1].

6.3 Entailment of Specifications and Internal Models

Once we have a description of the architecture and a set of specifications for the components, we are well on our way to proving that the system behavior will satisfy these

[1]Recall that in the geometric interpretation of a QSP, state constraints were n-dimensional polytopic prisms while temporal constraints are hyperplanes. The increased dimensionality adds complexity.

assertions. To do so, we will need one more piece: a model of the planner behavior for each of the components.

Ultimately it is necessary to connect the specifications together by a formal reasoning process. Abstract architectures and specification sets can create the structure to set up these types of proofs for each component by providing the child component specifications as axioms and the component's own specification as a goal. The process of showing that the specification is a logical consequence of the axioms for a non-trivial planner requires that we know something about how the planner itself will transform the problem that is not captured in the external structure. In a sense, we need to "peek inside the boxes" of the architecture.

This can be done in a number of ways. A deceptively simple one is to use the planner's implementation itself as a perfect-fidelity model. Unless the planner was already written with formal verification in mind, however, this will quickly lead to over-burdensome proofs and may make the entire analysis intractable. For most real-world systems a simplified model is necessary; it may even make certain assumptions about the behavior of the planner that are not formally verified, though these will naturally be areas of weakness for the proofs that rely on them. As an example, an activity planner that is often used as a coordinating planner may assert that the implementation only ever dispatches problems to subplanners that are free of state constraint interference. This simplifies the modeling of the *compose* operation at the cost of either an unchecked assumption, or the proof obligation to demonstrate that this property holds.

A good model of the system must provide the analyst with confidence that the fidelity is high enough to capture the system behavior accurately, yet be small enough to be analyzable within a reasonable time frame. It should not make more assumptions about the system than are absolutely necessary. To be useful with this framework, it should be built in a formal language that is powerful enough to express the specification logic, logically sound, and provides support to the programmer in creating their models and proofs. How to write these models is well beyond the scope of this thesis, but there is a substantial body of literature in the formal methods community that aims to convey this skillset to the motivated practitioner (c.f. [17] for a model-checking approach, [48] for a theorem proving one).

6.4 Example: Describing Sulu

The Sulu planner is a capable path planner that uses a disjunctive linear programming approach to solve finite horizon planning problems [8]. Sulu is a leaf planner, so we do not require a model of its internal transformations for architectural analysis. It produces grounded plans that use a fixed increment time discretization.

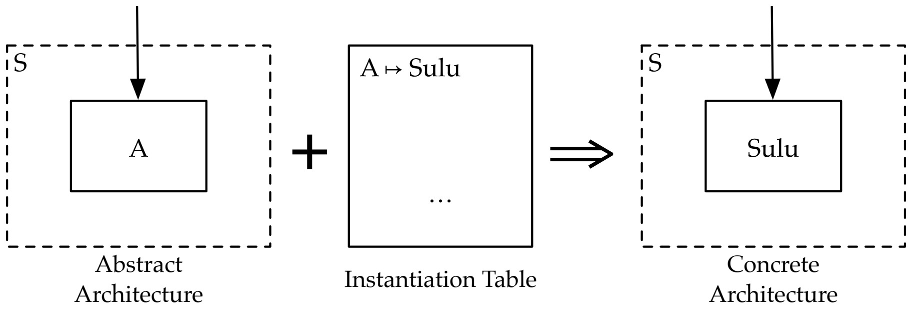

Figure 6-4: Abstract architecture, instantiation table, and concrete architecture for the leaf planner Sulu.

When constructing a description of the planner we should include these properties within the specification set. Representing grounding is simple, using the definition given above:

$$\forall mgp : QSP, s(m, g) = p \rightarrow \text{grounded}(p) \tag{6.4}$$

Exposing the time discretization within the logic requires creating a predicate that acts to select only those QSPs whose events are assigned to times that are multiples of the discretization interval. The predicate must have type $T \rightarrow T$ to be used within the specification logic, though it will need to be defined within the metalogic and made available via the specification logic's signature.

$$\text{discretized}(x) = \forall e : Event, e \in x \implies (t_e \bmod t_{int}) = 0 \tag{6.5}$$

Where t_e is the time assigned to the event and t_{int} is the discretization interval. After exposing this predicate within the specification logic signature, we can reference it as:

$$\forall mgp : QSP, s(m, g) = p \rightarrow \text{discretized}(p) \tag{6.6}$$

This property "bubbles up" to any coordinating planner that uses Sulu to generate sub-plans, causing its plans to be at least partially discretized. Capturing the discretization in this way allows the system analyst to determine if the sub-plans generated by Sulu are compatible with the rest of the system.

6.5 Example: Describing ScottyPath

In contrast to Sulu, ScottyPath is a continuous time planner that is well suited for long horizon planning but does not make use of the fine-grained dynamics that Sulu can handle. Used on its own, it is able to generate useful high level plans. We also conjecture that its fidelity can be increased by using its high-level plans to provide a road map that is then expanded by lower level planners (NB: this remains a conjecture, for which future work is planned). In this scenario, ScottyPath becomes a coordinating planner whose discretization depends on that of its child.

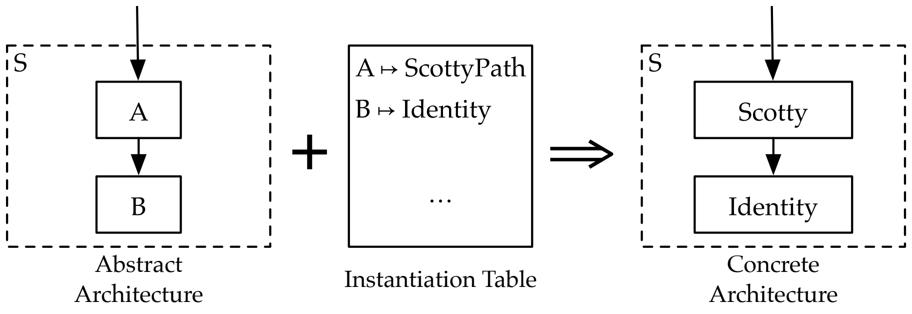

Figure 6-5: Abstract architecture, instantiation table, and concrete architecture for Scotty paired with an "identity" planner.

We can approximate the leaf planner behavior by binding the child to an "identity" planner, i.e. a planner that always returns its own input, as shown in figure 6-5. Notice that the composite, S, appears to the outside world as a leaf planner for all intents and purposes; it has no remaining assignments to be made. The specification for S then is the specification for a leaf-variant ScottyPath, including primary properties describing ScottyPath's particular environment representation. Rather than using a disjunction of half-planes to describe obstacles for the system to avoid, ScottyPath uses conjunctions of half-planes to describe obstacle-free safe regions that the system must remain within.

This environment representation, along with ScottyPath's requirement that all state and control variables be bounded, leads to a predicate on the state constraints defining the class of QSPs that maintain convex bounds on all variables at all times:

$$
\begin{aligned}
\text{bounded}(x) = \forall(v \in \text{vars}(x))&(e_s, e_f \in \text{events}(x)),\\
&|C(e_s, e_f)| > 0 \\
&\wedge \forall c \in C(e_s, e_f), e_s \leqslant t \leqslant e_f \implies l_c \leqslant v_t \leqslant u_c
\end{aligned}
\tag{6.7}
$$

ScottyPath will also require that the events in the QSP are sufficiently constrained to have a unique total ordering. We can model this by requiring that there exist a unique injection from the times of each event in the QSP onto the natural numbers that retains the ordering of the event timepoints.

$$
\begin{aligned}
\text{ordered}(x) = \exists!(f : \text{Event} \to \mathbb{N})&,\\
&\forall e\, e' \in \text{events}(x), (f(e) = f(e') \implies e = e') \\
&\wedge (f(e) \leqslant f(e') \implies t_e \leqslant t'_e)
\end{aligned}
\tag{6.8}
$$

Another architectural option is to use Sulu as the child planner. In this case, the input requirements to ScottyPath remain the same but the output properties change; namely, because Sulu discretizes time the portion of ScottyPath's output QSP that was formed by Sulu will also have a discrete time formulation. When Sulu is used to refine activities generated by Scotty, the QSPs dispatched to it will need to be flexible enough to allow Sulu to generate feasible plans. This is true for any arrangement of planners, but becomes more pressing when the planners make different assumptions about time or state.

Combining these predicates, we can describe ScottyPath in the specification logic. Here is a sentence showing the requirement for a total ordering and boundedness on state:

$$
\forall mgp : QSP, s(m, g) = p \to [(\neg\text{bounded}(m) \vee \neg\text{ordered}(g)) \to (p = \bot)] \tag{6.9}
$$

When ScottyPath is used as a coordinating planner, it may make sense to re-export the properties of the subplanner—referred to as t in the specification below—as properties of the composite system. To do so is simple, by stating that any witness object to a property asserted on solutions of the subplanner is also a witness to that property on the solutions of the composite:

$$\forall m\, m'\, g\, g'\, p\, p' : \mathrm{QSP}, [(t(m', g') = p') \to x] \to [(s(m, g) = p) \to x] \tag{6.10}$$

6.6 Example: Specification as a Design Aid

Description of existing components is a useful activity and provides the building blocks for new designs. However, we are not limited to discussing components that are already implemented—we can also design conceptual architectures that do not yet have an implementation and test them for feasibility. To do this we must first describe the abstract architecture and then determine if the components needed to instantiate it are possible to build.

To drive our example, let us consider the design of a least-commitment planning system. We can begin our design by writing the specification for the overall system, S. The specification set will begin with the following entries constraining S:

1. $\text{constraint\_correct}(s)$
2. $\forall m g p : T,\ s(m, g) \equiv p \to \text{least\_commit}(m, g, p)$

The basic abstract architecture has a single abstract component, A. We can attempt to instantiate this component with a single concrete component, if we have confidence that the problem can be fully solved using one program. If this is not the case—such as if the desired system needs to be able to operate over a very long time horizon—we may opt to break the system up into a coordinating planner A and a leaf planner B instead. A will inherit the overall system specification, as it is the root component. What specification is needed for the leaf planner?

To show that the combined system always yields a least commitment plan, we need to build a proof showing that the least_commit property holds given the subplanner specifications as assumptions. If we try to use a grounded planner like Sulu as our leaf planner, the assumption we obtain is that the plans coming from the subplanner will be completely determined with no choice left in them. For some combinations of models and goals this may be the least-commitment plan, but for the majority it will not be; constructing the proof of least commitment for the coordinating planner will almost certainly require us to ignore the plans generated by Sulu. This approach doesn't help us reduce the complexity of the problem in any meaningful way.

Another approach is to use a planner that is neither grounded nor least-commitment; a similar problem arises here, because we have no information about the amount of freedom that the planner will provide in its solutions to use in our proofs. Clearly, the best solution is to use a least-commitment subplanner. Our specification for the subplanner B thus mirrors the specification for S as a whole.

With a useful subplanner in place we can now decide what behavior we need to have from A, in terms of the subplans it dispatches to B, to satisfy its specification. Recall the commutative diagram from figure 6-3: subplanner problems are generated by a decompose operation and combined by a compose operation. Together, these operations must preserve the least_commit property for the specification to be met. Because the specification of least_commit is given as a meet of two objects, we want to select operations that preserve meets. If we assume compose $= \wedge$, then we want decompose to yield subproblems g_0, g_1 such that $p \equiv g_0 \wedge g_1$; then demonstrating least_commit is as simple as showing that the sentence

$$\forall mg, \text{decompose}(g) \equiv (g_0, g_1) \to [(m \wedge g_0) \wedge (m \wedge g_1)] \equiv (m \wedge g) \tag{6.11}$$

holds for the implementation of decompose.

Each of the steps taken in this example is an instance of a specification preserving refinement. Using this process we add more detail to the components in the architecture, their specifications, and their arrangement until we arrive at a set of implementable designs. If we have existing components we want to use, we can see their effect on the architecture by incorporating them and their specifications and attempting to prove the properties we want from the resulting system model. In some cases this will reveal a simple way to reuse existing components. In others, as outlined here, the existing component's properties are incompatible with our goals and we must design new ones. In either case the abstract architecture and its specification set enable the designer to evaluate options quickly.

6.7 Summary

Enterprise as a system provides planners to fit a number of different roles. These largely fall into two classes: leaf planners that have no children, and coordinator planners that do. The description of the input and output properties required for these planners can be expressed in the first-order specification logic defined in this thesis. In general, coordinator planners will tend to take on the characteristics of their subordinate planners. Two examples of planners, Sulu and ScottyPath, demonstrate different approaches to the representation of time and obstacles and example specification sentences were demonstrated. As a closing example, I showed how using the modeling approach described in this thesis can lead to insights about which components to use to achieve a given design objective, and how to begin a design by refining a specification.

Chapter 7

Conclusion

7.1 Summary of Contributions

In this thesis I have demonstrated the need for a formal modeling approach to aid in the design of composite planning systems. The existing literature has not yet solved this problem fully, and a major gap addressed in this thesis is the lack of an appropriate language for expressing specifications for the components of these systems. I have demonstrated that such a language can be created by using techniques from the field of mathematical logic, and that a strong semantics for a first-order logic can be created using the algebraic properties of the data structures used to represent both the problems solved by the components and the solutions themselves.

This language is intended for use in analyzing system architectures for correctness. To support this activity an appropriately minimal model of a planning system was demonstrated, and connected to specifications expressed in the logic. These models support a variety of abstraction levels and are able to express many different architectures; the distinction between an abstract architectural structure and the instantiation table used to make it concrete enables this.

In Chapter 6 I demonstrated the application of the logic to modeling problems, using *Enterprise* as a case study. *Enterprise* provides a wealth of planners designed for different applications. I have provided formal descriptions of the common properties of these planners in the specification logic, and addressed a few particular example planners (Sulu and ScottyPath) directly. As well as describing existing planners, Chapter 6 demonstrates how to use the logic as a design aid to guide the architect in creating systems to meet particular specifications, either by reusing existing planning components off-the-shelf or by creating new ones by refining their specifications to implementations.

7.2 Future Work

This thesis presents the beginning of what I expect to be a fruitful area of research at the intersection of software architecture, automated planning, and formal verification. I foresee three major areas of extension: the design and implementation of component models; extensions to the logical framework to enable more expressive specifications; and expansion of the architectural modeling techniques to relax the constraints imposed on the system design and allow more systems to be captured.

7.2.1 Component Modeling

Component models are the glue that binds together planner specifications. As discussed in Chapter 6, these models may be made at several levels of fidelity ranging from a full implementation to a highly abstracted approximation of the desired component behavior. Different levels of modeling are appropriate at different points in the design process, due to their cost to develop and computational requirements.

Extensions to this research should proceed with the approach adopted by the Mission Data System developed by Ingham, et al [16]. This would involve creating a library of pre-made models that can be used to explore various architectures, each of which is conceptually linked to a full implementation of the given algorithm for use in production systems. These models should be implemented in a computerized proof assistant or other formal modeling language so that they are able to be model-checked or used as the basis for theorem proving.

7.2.2 Logical Extensions

The logic defined in this paper is a reasonably powerful one, but it does not fully capture all of the properties that may be of interest in verifying an autonomous system. Some of these omissions are intentional—for instance, I do not attempt to model channel properties due to their treatment in the existing literature. Others are a matter of practicality given the need to demonstrate that such a logic could be defined in the first place.

A richer logic would give the system designer more powerful tools. Extending the logic to include functions, rather than simple function symbols, would enable component models to be expressed within the logic itself. Another approach that raises the logic to include linear operators would enable the modeling of finite computational resources such as time, memory, and risk. These are particularly important for real-time systems such as those used on spacecraft. A modal logic along the lines of LTL[1] would open up the world of partiality, allowing the system architect to consider the use of planners that may not return solutions, or to add bounds to their execution time. Logics that can represent infinite streams of data may be useful for modeling and verification of continuous planning and execution systems.

Most of these extensions involve finding (or defining) additional structure within the poset of problems beyond a Heyting algebra. Categorical semantics provides the theoretical framework for discovering these structures in terms of the category-theoretic analogs they must correspond to. For example, extending the logic to include functions means that the category of problems (say, of QSPs) must be *cartesian closed*; if sufficient structure can be defined to accomplish this, the category is a model of the simply-typed lambda calculus, which can then be used as the specification language. Similar statements can be made for the other extensions suggested in this section[2].

7.2.3 Flexible Architectural Models

In the introductory chapter of this thesis, I defined a restricted modeling formalism used to capture the major relations of a composite planning system's architecture. The approach is limited in that it forces the architecture to be a tree-like structure, and cannot represent systems in which planner have circular references to each other. This type of system, however, can be quite useful. A more expressive, graph-based representation would allow the modeler to capture more architectures and close the abstraction gap between the model and the as-built system.

[1]LTL, as a monadic logic, does not allow modeling of function symbols or relations that are not unary, which the logic defined in this thesis permits.

[2]c.f. Section 5.4.

Appendix A

Z3 Program to Find Pseudocomplements of a Heyting Algebra

```
(declare-datatypes () ((T O A B C AB AC BC ABC)))

(define-fun lte ((a T) (b T)) Bool
            (or
             (and (= a O) (= b O))
             (and (= a O) (= b A))
             (and (= a O) (= b B))
             (and (= a O) (= b C))
             (and (= a O) (= b AB))
             (and (= a O) (= b AC))
             (and (= a O) (= b BC))
             (and (= a O) (= b ABC))
             (and (= a A) (= b A))
             (and (= a A) (= b AB))
             (and (= a A) (= b AC))
             (and (= a A) (= b ABC))
             (and (= a B) (= b B))
```

```
                (and (= a B) (= b AB))
                (and (= a B) (= b BC))
                (and (= a B) (= b ABC))
                (and (= a C) (= b C))
                (and (= a C) (= b AC))
                (and (= a C) (= b BC))
                (and (= a C) (= b ABC))
                (and (= a AB) (= b AB))
                (and (= a AB) (= b ABC))
                (and (= a AC) (= b AC))
                (and (= a AC) (= b ABC))
                (and (= a BC) (= b BC))
                (and (= a BC) (= b ABC))
                (and (= a ABC) (= b ABC))
                ))

;; General properties of lte:
;; Transitivity
(assert (forall ((a T) (b T) (c T)) (implies (and (lte a b) (lte b c)) (lte a c))))
;; Reflexivity
(assert (forall ((a T)) (lte a a)))
;; Antisymmetry
(assert (forall ((a T) (b T)) (implies (and (lte a b) (lte b a)) (= a b))))

(check-sat)

;; Bounding instances
(assert (forall ((a T)) (lte O a)))
(assert (forall ((a T)) (implies (distinct a O) (not (lte a O)))))

(assert (forall ((a T)) (lte a ABC)))
(assert (forall ((a T)) (implies (distinct a ABC) (not (lte ABC a)))))
```

```
(define-fun is-meet ((a T) (b T) (c T)) Bool
            (and (and (lte c a) (lte c b))
                 (forall ((d T))
                         (implies (and (lte d a) (lte d b))
                                  (lte d c)))))

(define-fun is-join ((a T) (b T) (c T)) Bool
            (and (and (lte a c) (lte b c))
                 (forall ((d T))
                         (implies (and (lte a d) (lte b d))
                                  (lte c d)))))

;; Examples
(push)
(declare-const a T)
(declare-const b T)
(declare-const c T)
(assert (= a A))
(assert (= b B))

(push)

;; Find a meet between A and B: This is O.
(assert (= (is-meet a b c) true))

(check-sat)
(pop)

(push)

;; Find a join between A and B: This is AB.
```

```
(assert (= (is-join a b c) true))

(check-sat)
(pop)
(pop)

(push)
;; Theorem checking: all elements have a meet.
(echo "Meets (OK iff unsat):")
(assert (not (forall ((a T) (b T)) (exists ((c T)) (is-meet a b c)))))

(check-sat)
(pop)

(push)
;; Theorem checking: all elements have a join.
(echo "Joins (OK iff unsat):")
(assert (not (forall ((a T) (b T)) (exists ((c T)) (is-join a b c)))))

(check-sat)
(pop)

(declare-fun is-arrow (T T T) Bool)

;; Aliases for top and bottom
(define-fun Top () T ABC)
(define-fun Bot () T O)

;; Axioms of Heyting implication:
(define-fun self-implication () Bool
            (forall ((a T))
                    (is-arrow a a Top)))
```

```
(assert self-implication)

(define-fun modus-ponens () Bool
            (forall ((a T) (b T) (c T) (d T))
                    (implies (and (is-arrow a b c)
                                  (is-meet  a c d))
                                  (is-meet  a b d))))

(assert modus-ponens)

(define-fun arrow-elim () Bool
            (forall ((a T) (b T) (c T) (d T))
                    (implies (and (is-arrow a b c))
                             (is-meet  b c b))))

(assert arrow-elim)

(define-fun meets-distr () Bool
            (forall ((a T) (b T) (c T) (d T) (e T) (f T) (g T) (h T))
                    (implies (and
                              (is-meet  b c d)
                              (is-arrow a d e)
                              (is-arrow a b f)
                              (is-arrow a c g))
                             (is-meet  f g e))))

(assert meets-distr)

(define-fun all-implications () Bool
            (forall ((a T) (b T)) (exists ((c T)) (is-arrow a b c))))
```

```
(assert all-implications)


(define-fun bottom-implies-all () Bool
            (forall ((a T)) (is-arrow Bot a Top)))
(assert all-implications)


(check-sat)
(get-model)
```

Bibliography

[1] Ari K Jonsson, Paul H Morris, Nicola Muscettola, et al. "Planning in Interplanetary Space: Theory and Practice". In: *Proceedings of the Fifth International Conference on Artificial Intelligence Planning Systems* (Apr. 14, 2000–Apr. 17, 2000), p. 6.

[2] Douglas E. Bernard, Edward B. Gamble, Nicolas F. Rouquette, et al. *Remote Agent Experiment DS1 Technology Validation Report*. 2000.

[3] Nicola Muscettola, P Pandurang Nayak, Barney Pell, et al. "Remote Agent: To Boldly Go Where No AI System Has Gone Before". In: *Artificial Intelligence* 103.1-2 (Aug. 1, 1998), pp. 5–47. DOI: 10.1016/S0004-3702(98)00068-X. URL: http://linkinghub.elsevier.com/retrieve/pii/S000437029800068X.

[4] JPL. *JPL History: The 90s Overview*. May 5, 2016. URL: https://web.archive.org/web/20160505173534/https://www.jpl.nasa.gov/jplhistory/the90/ (visited on 02/21/2019).

[5] Ilana Dashevsky and Vicki Balzano. "JWST: Maximizing Efficiency and Minimizing Ground Systems". In: *2007 International Symposium on Reducing the Costs of Spacecraft Ground Systems and Operations Proceedings*. 2007, p. 6.

[6] Matt Luckcuck, Marie Farrell, Louise Dennis, et al. "Formal Specification and Verification of Autonomous Robotic Systems: A Survey". In: (June 29, 2018). arXiv: 1807.00048 [cs]. URL: http://arxiv.org/abs/1807.00048 (visited on 02/18/2019).

[7] Daniel Jackson. "Alloy: A Language and Tool for Exploring Software Designs". In: *(To Appear) Communications of the ACM* (2019).

[8] Thomas Léauté and Brian C Williams. "Coordinating Agile Systems through the Model-Based Execution of Temporal Plans." In: *AAAI* (Jan. 1, 2005). URL: https://dblp.org/rec/conf/aaai/LeauteW05.

[9] Enrique Fernández-González, Erez Karpas, and Brian Charles Williams. "Mixed Discrete-Continuous Planning with Convex Optimization." In: *AAAI* (Jan. 1, 2017). URL: https://dblp.org/rec/conf/aaai/Fernandez-Gonzalez17.

[10] Nancy G Leveson. *Engineering a Safer World*. Systems Thinking Applied to Safety. MIT Press, Jan. 13, 2012. 560 pp. ISBN: 0-262-29730-2.

[11] Todd Bayer. "Is MBSE Helping? Measuring Value on Europa Clipper". In: *2018 IEEE Aerospace Conference*. 2018 IEEE Aerospace Conference. Mar. 2018, pp. 1–13. DOI: 10/gfvsnh.

[12] *NASA Software Safety Guidebook*. NASA-GB-8719.13. National Aeronautics and Space Administration, Mar. 31, 2004. URL: https://standards.nasa.gov/file/2617/download?token=-ONrGarI.

[13] Tazeen Mahtab, Gregory T Sullivan, and Brian C Williams. “Automated Verification of Model-Based Programs Under Uncertainty”. In: (June 18, 2004), pp. 1–7. URL: https://groups.csail.mit.edu/mers/papers/isda04-mba-final.pdf.

[14] G. Brat, E. Denney, D. Giannakopoulou, et al. “Verification of Autonomous Systems for Space Applications”. In: *2006 IEEE Aerospace Conference*. 2006 IEEE Aerospace Conference. Mar. 2006, 11 pp.–. DOI: 10/bf3vxh.

[15] INCOSE Technical Operations. *INCOSE Systems Engineering Vision 2020*. INCOSE-TP-2004-004-02. International Council on Systems Engineering, 2007.

[16] Michel D Ingham, Robert D Rasmussen, Matthew B Bennett, et al. “Engineering Complex Embedded Systems with State Analysis and the Mission Data System”. In: *Journal of Aerospace Computing, Information, and Communication* 2.12 (Dec. 1, 2005), pp. 507–536. DOI: 10.2514/1.15265. URL: http://arc.aiaa.org/doi/10.2514/1.15265.

[17] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis Revised Edition*. MIT Press, Nov. 2011. 376 pp. ISBN: 978-0-262-01715-2.

[18] Benjamin Delaware, Clément Pit-Claudel, Jason Gross, et al. “Fiat”. In: *POPL '15*. The 42nd Annual ACM SIGPLAN-SIGACT Symposium. SIGPLAN, ACM Special Interest Group on Programming Languages, Jan. 1, 2015, pp. 689–700. ISBN: 978-1-4503-3300-9. DOI: 10.1145/2676726.2677006. URL: http://dl.acm.org/citation.cfm?doid=2676726.2677006.

[19] James McDonald and John Anton. *SPECWARE: Producing Software Correct by Construction*. Kestrel Institute, Mar. 14, 2001.

[20] M. Zimmerman, M. Rodriguez, B. Ingram, et al. “Making Formal Methods Practical”. In: *19th DASC. 19th Digital Avionics Systems Conference. Proceedings (Cat. No.00CH37126)*. 19th Digital Avionics System Conference. Proceedings. Vol. 1. Philadelphia, PA, USA: IEEE, 2000, 1B2/1–1B2/8. ISBN: 978-0-7803-6395-3. DOI: 10/fs55xh. URL: http://ieeexplore.ieee.org/document/886879/ (visited on 02/12/2019).

[21] K. Havelund, M. Lowry, and J. Penix. “Formal Analysis of a Space-Craft Controller Using SPIN”. In: *IEEE Transactions on Software Engineering* 27.8 (Aug./2001), pp. 749–765. ISSN: 00985589. DOI: 10.1109/32.940728. URL: http://ieeexplore.ieee.org/document/940728/ (visited on 05/21/2019).

[22] Klaus Havelund, Mike Lowry, SeungJoon Park, et al. “Formal Analysis of the Remote Agent Before and After Flight”. In: The Fifth NASA Langley Formal Methods Workshop. Virginia, June 2000, p. 12.

[23] Leslie Lamport. “Specifying Concurrent Systems with TLA+”. In: (Mar. 1999), p. 68.

[24] Klaus Havelund and Grigore Rou. “Runtime Verification - 17 Years Later”. In: *Runtime Verification*. Ed. by Christian Colombo and Martin Leucker. Vol. 11237. Cham: Springer International Publishing, 2018, pp. 3–17. ISBN: 978-3-030-03768-0 978-3-030-03769-7. DOI: 10.1007/978-3-030-03769-7_1. URL: http://link.springer.com/10.1007/978-3-030-03769-7_1 (visited on 05/21/2019).

[25] R. Halder, J. Proença, N. Macedo, et al. "Formal Verification of ROS-Based Robotic Applications Using Timed-Automata". In: *2017 IEEE/ACM 5th International FME Workshop on Formal Methods in Software Engineering (FormaliSE)*. 2017 IEEE/ACM 5th International FME Workshop on Formal Methods in Software Engineering (FormaliSE). May 2017, pp. 44–50. DOI: 10/gfvnnp.

[26] M Ono, Brian C Williams, and Lars Blackmore. "Probabilistic Planning for Continuous Dynamic Systems under Bounded Risk". In: *Journal of Artificial Intelligence Research* 46 (Jan. 1, 2013), pp. 511–577. DOI: 10.1613/jair.3893. URL: https://jair.org/index.php/jair/article/view/10808.

[27] Catharine L R McGhan, Richard M Murray, Romain Serra, et al. "A Risk-Aware Architecture for Resilient Spacecraft Operations". In: *2015 IEEE Aerospace Conference*. 2015 IEEE Aerospace Conference. IEEE, Jan. 1, 2015, pp. 1–15. ISBN: 978-1-4799-5379-0. DOI: 10.1109/aero.2015.7119035. URL: http://ieeexplore.ieee.org/document/7119035/.

[28] Phil Kim, Brian C Williams, and Mark Abramson. "Executing Reactive, Model-Based Programs through Graph-Based Temporal Planning". In: *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence*. IJCAI 2001. Seattle, Washington, Jan. 2001, p. 7.

[29] Benjamin James Ayton. "Risk-Bounded Autonomous Information Gathering for Localization of Phenomena in Hazardous Environments". Master's Thesis. Cambridge, Massachusetts: Massachusetts Institute of Technology, Sept. 2017. 150 pp.

[30] NASA. *Planetary Science and Technology Through Analog Research (PSTAR) Program Abstracts of Selected Proposals*. NNH15ZDA001N-PSTAR. National Aeronautics and Space Administration, 2016. URL: https://astrobiology.nasa.gov/research/astrobiology-at-nasa/pstar/ (visited on 04/19/2019).

[31] Rina Dechter, Itay Meiri, and Judea Pearl. "Temporal Constraint Networks". In: *Artificial Intelligence* 49.1-3 (May 1, 1991), pp. 61–95. DOI: 10.1016/0004-3702(91)90006-6. URL: http://linkinghub.elsevier.com/retrieve/pii/0004370291900066.

[32] Henri Mühle. "A Heyting Algebra on Dyck Paths of Type A and B". In: *Order* 34.2 (July 2017), pp. 327–348. ISSN: 0167-8094, 1572-9273. DOI: 10/gfxzhv. arXiv: 1312.0551. URL: http://arxiv.org/abs/1312.0551 (visited on 04/04/2019).

[33] Sten Lindström and Erik Palmgren. "Introduction: The Three Foundational Programmes". In: *Logicism, Intuitionism, and Formalism*. Ed. by Sten Lindström, Erik Palmgren, Krister Segerberg, et al. Dordrecht: Springer Netherlands, 2009, pp. 1–23. ISBN: 978-1-4020-8925-1 978-1-4020-8926-8. DOI: 10.1007/978-1-4020-8926-8_1. URL: http://link.springer.com/10.1007/978-1-4020-8926-8_1 (visited on 04/08/2019).

[34] J Baez and M Stay. "Physics, Topology, Logic and Computation: A Rosetta Stone". In: *New Structures for Physics*. Vol. 813. Chapter 2 vols. Berlin, Heidelberg: Springer, Berlin, Heidelberg, Jan. 1, 2010, pp. 95–172. ISBN: 978-3-642-12820-2. DOI: 10.1007/978-3-642-12821-9_2. URL: https://link.springer.com/chapter/10.1007/978-3-642-12821-9_2.

[35] A. S. Troelstra. "History of Constructivism in the 20th Century". In: *Set Theory, Arithmetic, and Foundations of Mathematics*. Ed. by Juliette Kennedy and Roman Kossak. Cambridge: Cambridge University Press, 2011, pp. 150–179. ISBN: 978-0-511-91061-6. DOI: 10.1017/CBO9780511910616.009. URL: https://www.cambridge.org/core/product/identifier/CBO9780511910616A014/type/book_part (visited on 04/08/2019).

[36] John Myhill. "Embedding Classical Logic in Intuitionistic Logic". In: *Zeitschrift für Mathematische Logik und Grundlagen der Mathematik* 19.3-6 (1973), pp. 93–96. ISSN: 00443050, 15213870. DOI: 10/bdg42n. URL: http://doi.wiley.com/10.1002/malq.19730190307 (visited on 03/19/2019).

[37] P T Johnstone. *Sketches of an Elephant: A Topos Theory Compendium*. Oxford University Press, Sept. 12, 2002. 716 pp. ISBN: 978-0-19-851598-2. URL: http://books.google.com/books?id=TLHfQPHNs0QC&printsec=frontcover&dq=inauthor:johnstone+elephant&hl=&cd=1&source=gbs_api.

[38] Martin Davis. "The Incompleteness Theorem". In: *Notices of the AMS* 53.4 (2006), p. 5.

[39] Arend Heyting. *Die formalen Regeln der intuitionistischen Logik*. Sitzungsberichte der Preussischen Akademie der Wissenschaften. Physikalisch-mathematische Klasse. 2. Berlin, 1930.

[40] Hui X. Li. "Kongming: A Generative Planner for Hybrid Systems with Temporally Extended Goals". PhD Thesis. Massachusetts Institute of Technology, June 23, 2010.

[41] E. Bradley and F. Zhao. "Phase-Space Control System Design". In: *IEEE Symposium on Computer-Aided Control System Design*. IEEE Symposium on Computer-Aided Control System Design. Mar. 1992, pp. 68–75. DOI: 10/d35xn2.

[42] Andreas G Hofmann. "Robust Execution of Bipedal Walking Tasks From Biomechanical Principles". PhD Thesis. Massachusetts Institute of Technology, 2006.

[43] David Wang and Brian C Williams. "tBurton: A Divide and Conquer Temporal Planner." In: *aaai.org* (). URL: http://www.aaai.org/ocs/index.php/AAAI/AAAI15/paper/viewFile/9749/9772.

[44] Heinz-Dieter Ebbinghaus and Jörg Flum. *Finite Model Theory*. OCLC: 604227793. Berlin; New York: Springer, 1995. ISBN: 978-3-662-03182-7 978-3-662-03184-1. URL: http://public.eblib.com/choice/publicfullrecord.aspx?p=3097142 (visited on 05/03/2019).

[45] Erik Palmgren. "Constructive Mathematics: Course Notes for the Copenhagen Summer School in Logic". Copenhagen, NL, 1997.

[46] Douglas R Smith and Stephen J Westfold. *Toward the Synthesis of Constraint Solvers*. Palo Alto, CA, Nov. 2, 2013, pp. 1–45. URL: https://www.kestrel.edu/home/people/smith/pub/CW-report.pdf.

[47] Vincent Danos and Roberto Di Cosmo. *Introduction to Linear Logic: Notes for the MPRI Course*. 2016. URL: http://www.dicosmo.org/CourseNotes/LinLog/IntroductionLinearLogic.pdf (visited on 04/26/2019).

[48] Adam Chlipala. *Certified Programming with Dependent Types*. MIT Press, Dec. 2013. ISBN: 978-0-262-02665-9.