Timed Model-based Programming: Executable Specifications for Robust Mission-Critical Sequences

by

Michel Donald Ingham

B.Eng., McGill University (1995)
S.M., Massachusetts Institute of Technology (1998)

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

Doctor of Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2003

Author .....
Department of Aeronautics and Astronautics
May 23, 2003

Certified by .....
Brian C. Williams
Associate Professor of Aeronautics and Astronautics
Committee Chairman

Certified by .....
David W. Miller
Associate Professor of Aeronautics and Astronautics

Certified by .....
Eric Feron
Associate Professor of Aeronautics and Astronautics

Accepted by .....
Edward M. Greitzer
H.N. Slater Professor of Aeronautics and Astronautics
Chair, Department Committee on Graduate Students

Timed Model-based Programming: Executable Specifications for Robust Mission-Critical Sequences

by

Michel Donald Ingham

Submitted to the Department of Aeronautics and Astronautics
on May 23, 2003, in partial fulfillment of the
requirements for the degree of
Doctor of Science

Abstract

There is growing demand for high-reliability embedded systems that operate robustly and autonomously in the presence of tight real-time constraints. For robotic spacecraft, robust plan execution is essential during time-critical mission sequences, due to the very short time available for recovery from anomalies. Traditional approaches to encoding these sequences can lead to brittle behavior under off-nominal execution conditions, due to the high level of complexity in the control specification required to manage the complex spacecraft system interactions. This work describes *timed model-based programming*, a novel approach for encoding and robustly executing mission-critical spacecraft sequences.

The timed model-based programming approach addresses the issues of sequence complexity and unanticipated low-level system interactions by allowing control programs to directly read or write "hidden" states of the plant, that is, states that are not directly observable or controllable. It is then the responsibility of the program's execution kernel to map between hidden states and the plant sensors and control variables. This mapping is performed automatically by a *deductive controller* using a common-sense plant model, freeing the programmer from the error-prone process of reasoning through a complex set of interactions under a range of possible failure situations.

Time is central to the execution of mission-critical sequences; a robust executive must consider time in its control and behavior models, in addition to reactively managing complexity. In timed model-based programming, control programs express goals and constraints in terms of both system state and time. Plant models capture the underlying behavior of the system components, including nominal and off-nominal modes, probabilistic transitions, and timed effects such as state transition latency.

The contributions of this work are threefold. First, a semantic specification of the timed model-based programming approach is provided. The execution semantics of a timed model-based program are defined in terms of legal state evolutions of a physical

plant, represented as a factored Partially Observable Semi-Markov Decision Process. The second contribution is the definition of graphical and textual languages for encoding timed control programs and plant models. The adoption of a visual programming paradigm allows timed model-based programs to be specified and readily inspected by the systems engineers in charge of designing the mission-critical sequences. The third contribution is the development of a Timed Model-based Executive, which takes as input a timed control program and executes it, using timed plant models to track states, diagnose faults and generate control actions. The Timed Model-based Executive has been implemented and demonstrated on a representative spacecraft scenario for Mars entry, descent and landing.

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

Acknowledgments

First and foremost, I dedicate this thesis to the love of my life, my amazing wife Leslie. Words cannot express how grateful I am to you for being my sunshine through this long, arduous and sometimes stressful process. Thank you for your endless patience and constant support. I couldn't have done it without you!

To the rest of my family, Mom, Dad, Scott, and Diana, thanks for always being there with a kind word of encouragement when I needed it. It's been a long time since I've had a completely guilt-free family vacation, but now that I'm done, this is one of the things I look forward to most.

To my advisor Brian, thank you for providing me with the opportunity to grow as a researcher and engineer over the past five years as your doctoral student. I've benefitted greatly from your guidance and shared insight, as well as from my interactions with the outstanding team you've assembled in the MERS group.

To my doctoral committee members, Dave Miller, Eric Feron, and Ed Crawley, and my thesis readers, Dave Watson and Kristina Lundqvist, thank you for your support and positive feedback.

To Lorraine Fesq, a very special expression of thanks for all the help you've provided to me over the last few years - I've learned so much from working with you. Thanks for all the positive reinforcement, and your unwavering confidence in me. Most of all, thanks for caring. The world needs more folks like you.

To Margaret Yoon, I am indebted to you for all the assistance you've provided me over the last year. It's been a tremendous stress reliever to know that I can count on you to make things happen. You've been a godsend to the MERS team - I don't know what we ever did before you joined us!

To SharonLeah Brown and the rest of the MIT Space Systems Lab staff, I thank you all for your contributions in helping to make the Lab a home away from home for me and my labmates. Thanks also to Marie Stuppard and the Aero/Astro Department staff, for their unfailing helpfulness and patience in dealing with stressed-out graduate students.

To all the folks at JPL who have gone out of their way to make me feel like one of the team, thanks so much. In particular, I owe a huge thank-you to Steve Chien, Tony Barrett and the rest of the AI Group for taking me in upon my transition to the west coast. Another big thanks goes out to the MDS team, for being such a pleasure to work with, particularly Dan Dvorak, Bob Rasmussen, Sandy Krasner, Kenny Meyer, Nic Rouquette, George Rinker, and Alex Moncada. And thanks to Ben Smith and others who provided me with some great feedback in preparation for my thesis defense.

To Milind Tambe and the Teamcore group at USC/ISI, thanks again for finding a place for me while I was in the Green Card "holding pattern". I very much enjoyed getting to know all of you during my brief stay at ISI. Your group made me feel very welcome.

To Mike Pekala and the rest of the team at JHU/APL, thanks for all your hard work, frequently above and beyond the call of duty. It's been great to have you guys as part of the team.

To the best friends I've made in my years here in Cambridge: Brett deBlonk, Greg Mallory, Seung Chung, Alice Liu, Alex Makarenko, Fred Bourgault, Yool Kim, and John & Sarah Enright, you guys are the best! I will always cherish the memories from the good old days at MIT.

To Paul Elliott, thanks for your invaluable contributions in the implementation of Titan.

To Greg Sullivan, thank you for sharing your insights on my research work, and for stepping in to unload me of some extra responsibilities toward the end of my doctoral program.

To all the other students at MIT whom I've had the pleasure of working with, "merci beaucoup" and best of luck to you in your future endeavors.

And last but definitely not least, all my love and appreciation goes out to the rest of my family, as well as the Ravestein family.

Finally, the acknowledgments wouldn't be complete without recognizing the sponsorship of this research by NASA's Cross Enterprise Technology Development Pro-

gram, under contract NAG2-1466. I also acknowledge the Northrop Grumman Space Technology company for giving me permission to include the Chandra spacecraft activation sequence flowchart in Appendix A. Copyright of this flowchart belongs to Northrop Grumman Space Technology.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

There is growing demand for high-reliability embedded systems that operate robustly and autonomously in the presence of tight real-time constraints and high levels of uncertainty. The ever-increasing complexity of such systems imposes stringent requirements on execution technology in the areas of software verifiability, temporal reactivity and fault management. Through advances in embedded software, the risks associated with high levels of system complexity can be mitigated. Robotic space exploration provides an interesting domain for the study of these embedded programming issues.

1.1 Robustness in Mission-Critical Sequences

In the past, high levels of robustness under extreme uncertainty was largely the realm of deep space exploration. Billion-dollar space systems, like the Galileo spacecraft, have achieved robustness by employing sizable software development teams and by using many operations personnel to handle unforeseen circumstances as they arise. Efforts to make these missions highly capable at reduced costs have proven challenging, producing notable losses, such as the Mars Polar Lander and Mars Climate Orbiter failures [92]. Contributing to these failures was the difficulty associated with thinking through the large space of potential interactions between the embedded software and its underlying hardware, and writing flight code to handle all possible situations.

The traditional approach to controlling spacecraft is through nominal command sequences, which are time-tagged lists of commands and macros. These sequences specify actions down to the level of detailed hardware commands, whose effects can potentially be felt across spacecraft subsystem boundaries. Thus, to build confidence that the spacecraft will behave as predicted, engineers must perform careful modeling and extensive testing of these sequences on sophisticated hardware-in-the-loop simulation testbeds, as well as the flight hardware itself, prior to launch. Once uploaded to the onboard flight computer, these sequences are executed by simply issuing the appropriate commands at their specified times, in open-loop fashion [28].

For flight activities where more flexible event-driven execution is necessary, such time-triggered sequences are insufficient due to their inability to represent conditional response. Thus, timed command sequences can be augmented with rule-based engines [80] or hard-coded state machines [71] running as concurrent processes, which periodically check the available onboard measurements for satisfaction of a trigger condition and issue predetermined commands or macros in response. These conditional execution mechanisms are also used for onboard fault protection. Off-nominal behavior is usually handled by putting the spacecraft into "safe mode," in which all non-essential spacecraft functions are disabled. Once in safe mode, the spacecraft communicates its status to the ground controllers and then waits for them to diagnose the problem and uplink corrective actions. These interventions can be costly, both in terms of ground operations costs and the science opportunities lost while in safe mode.

During mission-critical activities, such as planetary fly-bys, orbital insertion and entry, descent and landing, putting a spacecraft into safe mode would result in loss of the mission. In such situations, standard fault responses are usually disabled, and fault protection is provided by highly specialized dedicated sequences. These flight software modules tend to be significantly more complex than non-critical sequences, due to the need for the sequence to cover a broad set of possible fault scenarios and provide "fly-through-failure" capability. Generating and testing these critical sequences is an extremely expensive process; this cost can dominate the mission

operations budget even though these sequences represent a small fraction of the overall mission duration [29]. Furthermore, at execution time, the complexity problem is exacerbated by the very short time available for recovery from anomalies, and can result in "brittle," non-robust behavior in unexpected off-nominal conditions.

For example, consider the leading hypothesis for the cause of the Mars Polar Lander failure [13]. Mars Polar Lander used a set of Hall effect sensors in its legs to detect touchdown. These sensors were watched by a set of software monitors that were designed to turn off the engine upon landing. As the lander descended into the Mars atmosphere, it deployed its legs. At this point it is most likely that the force of deployment produced a noise spike on the leg sensors, which was latched by the software monitors. The lander continued to descend, using a laser altimeter to detect distance to the surface. At an altitude of approximately 40 m, the lander began polling its leg monitors to determine touchdown. It would have immediately read the latched noise spike and shut down its engine prematurely, resulting in the spacecraft plummeting to the surface from 40 m.

If the spacecraft had been programmed with the ability to combine information from multiple sensors and reason about its state, it would have recognized the conflicting information being provided by the altimeter sensor (which indicated 40 m altitude) and the touchdown sensor (which indicated that touchdown had already occurred). This kind of inconsistency would have led the spacecraft to the conclusion that a sensor problem was most likely to blame. The desired response would have been to gather additional information to determine the correct altitude and choose a conservative approach in the presence of uncertainty.

Embedding this type of reasoning into a spacecraft based on current flight software practices is a challenging and costly process. The full space of potential failures and their interactions with the embedded software is too large for programmers to completely enumerate within a dedicated landing sequence encoded as a complex, ad-hoc software module. The cost of generating this type of specialized software and testing it against a sufficient set of simulated failure scenarios is at odds with the "faster-better-cheaper" philosophy adopted in missions like Mars Polar Lander.

The next generation of robotic explorer spacecraft will need to be endowed with unprecedented levels of "self-awareness," enabled through onboard goal-driven commanding, state-based control and fault management built into the nominal execution loop. Achieving this objective at a reasonable cost requires a rethinking of traditional embedded flight software architectures, in which fault protection is considered an "add-on" capability [22, 69]. This work describes a novel approach for encoding and executing robust mission-critical spacecraft sequences.

1.2 Role of Time in Mission-Critical Sequences

Given the urgency associated with mission-critical sequences, it is clear that time should be a central consideration during their execution. While managing the complexity of state-of-the-art spacecraft systems at reactive time scales, as discussed in the previous section, a robust onboard executive must consider time in its control and behavior models.

Time-critical spacecraft sequences generally include hard-coded delays between certain actions, which implicitly capture knowledge about the state of the spacecraft or its environment. For example, in the onboard sequence for atmospheric descent of a Mars lander spacecraft, a delay of approximately 10 seconds between jettison of the lander's heatshield and deployment of its legs is introduced, to ensure that the deploying legs do not impact the separating heatshield. Engineers choose to encode this type of engineering knowledge implicitly via a timing constraint, rather than explicitly including the relevant states in the plant model, either because it simplifies the onboard reasoning, or because of system observability limitations. For example, the heatshield separation distance from the lander is not explicitly measurable, though its expected behavior envelope has been determined from pre-launch empirical testing and statistical simulation. Robust control programs for time-critical sequences must allow for specification of such timing constraints, in addition to reactively handling the complexity problem.

In addition to its aforementioned use in sequences to represent unmodeled states,

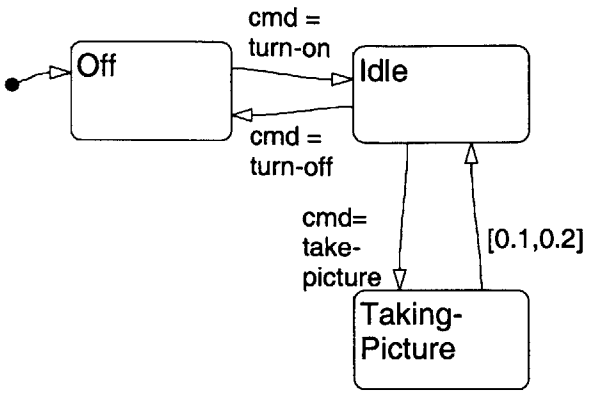

Figure 1-1: State transition diagram depicting nominal operating modes of a simple science camera and transitions between modes.

time also plays a role within the underlying system models that capture the behavior of spacecraft hardware and software in various nominal and off-nominal modes of operation. In order to realistically describe certain component behaviors, such as mode transition latency and gradual state evolution, the models of these components must allow for representation of these timed effects. For example, consider the nominal sequence of mode transitions of a simplified science camera instrument. The behavior of the camera is represented by the state transition diagram in Figure 1-1. From its "off" mode, the camera can be turned on by issuing it a single "turn-on" command, which puts it into "idle" mode. The transition from "off" to "idle" can be assumed to occur instantaneously, i.e. it takes no time. In order to capture an image with the camera, the "take-picture" command is issued. This results in the shutter of the camera opening for between one and two tenths of a second ("taking-picture" mode). This type of non-deterministic transient behavior is representative of the inherent uncertainty associated with a real hardware component, and is depicted in the figure by the state transition from "taking-picture" back to "idle" mode, labeled [0.1,0.2].

Traditional methods of controlling embedded systems generally assume synchronous Markovian behavior, where all component state transitions occur simultaneously at discrete instants, and a component's next state depends only on the current state and the action taken. However, when multiple components with different timed behaviors are composed in a system model, the Markov assumption does not apply.

For such systems, an asynchronous semi-Markov model becomes appropriate. In this model, time is modeled as a continuous variable, though the controller receives observations and makes decisions only at discrete points [58, 66]. A component's next state depends on its current state, as well as the amount of time spent in its current state; actions (such as "taking-picture") can persist over time periods with non-deterministic length. Given a semi-Markov system, time must be explicitly considered in the reasoning process.

1.3 Timed Model-based Programming

The objective of this work is to address the problems described in the previous two sections through *timed model-based programming (TMBP)*, a novel approach for encoding and executing robust mission-critical spacecraft sequences. This approach aims to avoid common-sense mistakes resulting from sequence complexity and unanticipated low-level system interactions. This is accomplished by adopting state-based control specifications and by automatically reasoning through probabilistic, timed models of nominal and off-nominal plant behavior, in order to track the system's state and deduce appropriate control actions.

TMBP addresses the challenge of managing the complexity of a timed system in two ways. First, it leverages and extends a new class of intelligent embedded systems that automatically diagnose and plan courses of action at reactive time scales, based on models of themselves and their environment [15, 43, 48, 83, 85]. This paradigm, called *model-based autonomy* [84], has been demonstrated in space on the NASA Deep Space One (DS-1) spacecraft [4], and on several subsequent space systems [30, 44]. Second, it elevates the level at which an engineer programs through a graphical specification language that is directly executable. This language allows the programmer to delegate, to the language's compiler and run-time execution kernel, tasks involving reasoning through system interactions, such as low-level commanding, monitoring, diagnosis, and repair.

The TMBP approach described in this thesis is an evolution of the model-based

programming paradigm, introduced in [86] and formally presented in [90]. A model-based program is composed of two parts. The first is a *control program*, which uses standard embedded programming constructs, like iteration, conditional branching, parallel and sequential composition, and preemption, to codify specifications of desired system state evolution. In addition, to execute the control program, the execution kernel needs a model of the system it must control. Hence, the second part is a *plant model*, which captures the physical plant's nominal behavior and common failure modes. This model unifies constraints, concurrency and Markov processes.

The model-based programming approach provides the following key features:

1. *state-based specification* – In the model-based programming paradigm, control programs for embedded systems are written by specifying desired state trajectories of the plant. Unlike other embedded programming languages, like Esterel [6], which interact with a physical plant by reading sensors and setting control variables (left, Figure 1-2), a model-based programming language allows the programmer to interact directly with "hidden" plant states, that is, states that are not directly observable or controllable (right, Figure 1-2). Since engineers prefer to reason about embedded systems in terms of state evolutions, state-based specifications provide a natural means of encoding these systems. Furthermore, because the details of how states are estimated and achieved are omitted, state-based control programs are less complex than traditional embedded programs, allowing for easier verification by systems engineers.

2. *executable specification* – As mentioned above, control programs operate on, and are conditioned on, system states. Execution of a control program requires mapping from the state goals specified in the program to actuator commands that achieve the goals, and from the sensor observations to the current system state. In the model-based programming approach, this mapping between states and sensors/actuators is performed automatically, by a *deductive controller* that reasons through a common-sense *plant model*. This model represents the set of possible behaviors of the system components and the set of interactions be-

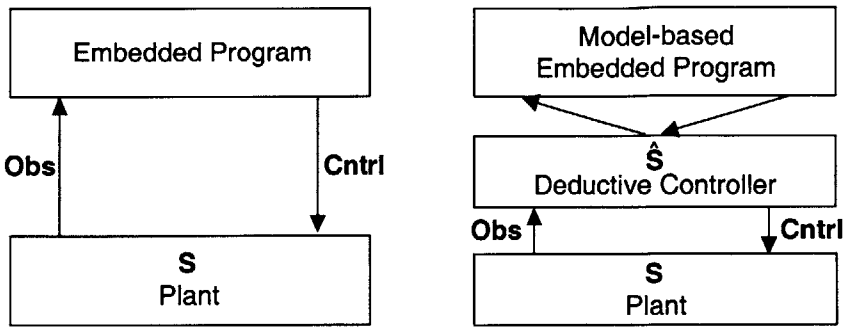

Figure 1-2: Model of interaction for traditional embedded languages (left) and model-based programming languages (right). Traditional embedded languages interact with a physical plant by reading sensors and setting control variables. Model-based programming languages interact directly with "hidden" plant states.

tween components. The deductive controller's on-line reasoning capabilities relieve the programmer of the responsibility of *a priori* encoding the complex set of low-level system interactions, for all possible execution scenarios. The resulting control program provides a simpler specification of desired system behavior, which is directly executable by the model-based programming language's execution kernel.

3. *fault-aware execution* – Model-based programs must ensure correct synthesis of behavior in the presence of failures. In addition to representing the nominal modes of operation of the system components, the plant model captures off-nominal behaviors as well. The deductive controller performs model-based diagnosis and reactive planning to enable the executive to detect and respond to failures on-the-fly, within the state-achievement loop. These failure recoveries are executed in a manner that is transparent to the control program. In addition, conditional reactions to unrepairable fault states can be encoded directly into the control program, resulting in a system that is more robust to onboard failures and anomalies than traditional procedural executives.

Though model-based programming has proven to be an effective approach for encoding robust spacecraft sequences for a variety of mission scenarios and spacecraft designs [25, 44], it does not provide the ability to capture the types of timed

behaviors described in Section 1.2. For example, a model-based program's control specification cannot represent the time delay between heatshield jettison and leg deployment, which captures the need to wait until the (unobservable) distance from the separating heatshield is large enough to avoid damage to the deploying legs. Furthermore, a model-based program cannot represent, in its plant behavior specification, the type of non-deterministic timed transition exhibited by the camera component (see Figure 1-1). In order to address these limitations, the TMBP approach extends model-based programming with the following additional key feature:

4. *timed specification* – The model-based programming paradigm is augmented by introducing clock variables and timing constraints, at both the control program and the plant model levels. Introducing time into the control programs allows their execution to be dependent on time as well as system state. In this approach, clock states are treated just like any other state, implying that *timed control programs* can set and read clock variables just as they set and read plant state variables. At the level of the *timed plant models*, introducing clock variables allows the representation of inherently asynchronous timed behaviors, such as transient states and transition latencies. Thus, TMBP generalizes the Markovian plant models defined in untimed model-based programming to accommodate semi-Markov plant behavior. These augmentations to the model-based programming approach provide the expressivity necessary for time-critical real-time embedded applications.

Another contribution of this thesis is the development of a visual programming paradigm for model-based programs (both timed and untimed). This contribution maps to the fifth key feature of TMBP:

5. *visual specification* – Engineers generally prefer to use visual representations of system specifications over textual encodings. For this reason, StateCharts [37] and similar formalisms have become fairly standard tools in the design and analysis of embedded systems. Timed model-based programming provides both a textual and a visual programming paradigm. Graphical languages are used

to encode both the control programs and the plant models. The graphical language used to specify timed control programs is a compact hierarchical state-based formalism, in the spirit of Timed StateCharts [47], called *Timed Hierarchical Constraint Automata (THCA)*. Timed plant models are specified using another graphical formalism, called *Timed Concurrent Constraint Automata (TCCA)*, which represents physical component behavior through nominal and faulty component modes, constraints and probabilistic timed transitions. The adoption of a visual encoding for timed model-based programs allows them to be specified and readily inspected by the systems engineers in charge of designing mission-critical sequences. The alternative textual language used to encode timed model-based programs is called the *Reactive Model-based Programming Language (RMPL)*. The RMPL constructs needed to encode untimed control programs have been previously introduced in [43, 86, 90]. The extensions to RMPL required to express untimed plant models have been presented in [87].

Later sections of this thesis describe how these features are provided by timed model-based programs and introduce an implemented executive that interprets and executes these programs at run-time.

1.4 Timed Model-based Execution

Figure 1-3 presents the TMBP architecture. A timed model-based program is executed by automatically generating a control sequence that moves the physical plant to the states specified by the timed control program. These specified states are called *configuration goals*. Program execution is performed by a *Timed Model-based Executive*. Similar to the model-based executives that operate on untimed model-based programs, a Timed Model-based Executive consists of two modules, a *control sequencer* and a *deductive controller*. The control sequencer is responsible for generating the sequence of configuration goals prescribed by the control program. Each configuration goal specifies an abstract state for the plant to achieve. The deductive controller is responsible for estimating the most likely current state based on observations from

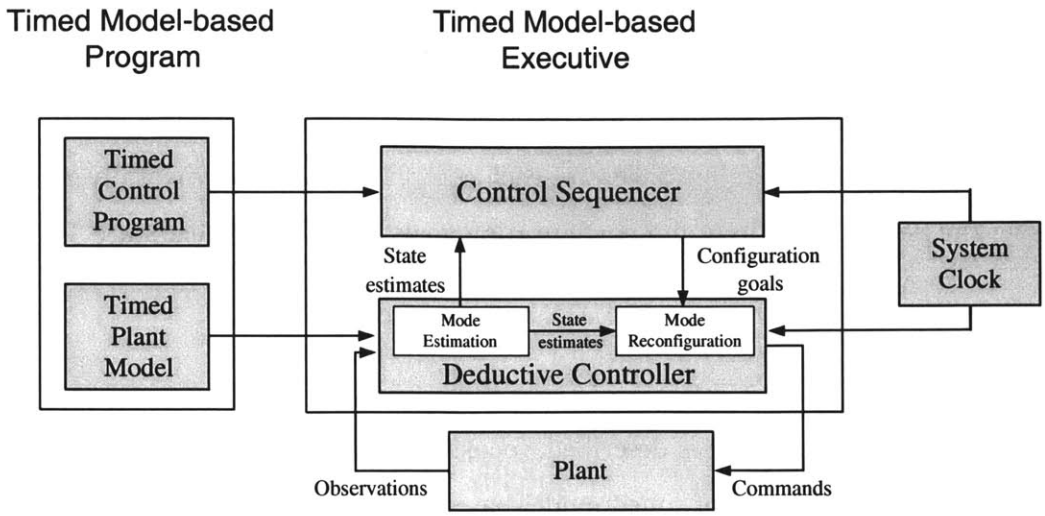

Figure 1-3: Timed model-based programming architecture. The Timed Model-based Executive automatically generates a sequence of commands that moves the physical plant to a state that satisfies the configuration goal specified by the control program, based on the current state estimate and system time.

the plant (*mode estimation*), and for issuing commands to move the plant through a sequence of states that achieve the configuration goals (*mode reconfiguration*) [83, 85].

The TMBP architecture extends the model-based programming architecture described in [90], by allowing both the control sequencer and the deductive controller to access the current time from the *system clock*. The control sequencer can thus execute control programs conditioned on time constraints, which are resolved based on input from the system clock, and state constraints, which are resolved based on state estimates deduced from the timed plant model.

The deductive controller's mode estimation capability continually estimates the most likely state of the plant. Given the latest observations from the plant sensors and the current time from the system clock, mode estimation reasons through the timed behaviors of the physical plant, to confirm the successful execution of commands and the achievement of configuration goals, and to diagnose failures. The deductive controller's mode reconfiguration capability continually tries to transition the plant towards a target state that satisfies the configuration goals, while maximizing some reward metric at that target. When the plant strays from the specified goals due to

failures, the deductive controller analyzes sensor data to identify the current state of the plant, and then moves the plant to a new state that, once again, achieves the desired goals. The executive is reactive in the sense that it responds immediately to changes in goals and to failures; that is, each control action is incrementally generated using the new observations and configuration goals provided in each state.

1.5 Outline

In Chapter 2, related work is described, focusing on fields of research which have influenced the development of timed model-based programming, and comparisons with other paradigms that address similar problems. Discussion of the timed model-based programming approach begins in Chapter 3 with a motivating example of a timed model-based program and its execution. The semantics of timed model-based program execution are then described in Chapter 4. The next two chapters describe the implementation of the two main parts of the Timed Model-based Executive: Chapter 5 focuses on the control sequencer and the textual and graphical languages used to write control programs, while Chapter 6 focuses on the deductive controller and the graphical language used to specify plant models. The implementation details of the integrated executive are discussed in Chapter 7, which also presents a demonstration of the executive on a Mars entry, descent and landing scenario. Finally, Chapter 8 concludes the thesis with a summary of the contributions and a discussion of directions for future work.

Chapter 2

Related Work

The TMBP approach unifies concepts from synchronous and concurrent constraint programming languages, robotic execution languages, model-based programming and execution, and formal modeling of real-time systems. This chapter discusses each of these fields of research, identifying the concepts that TMBP shares with each of them. In addition, it addresses related work on another real-time embedded architecture, the Mission Data System, under development at NASA's Jet Propulsion Laboratory (JPL), as an example of a project that shares many common themes with TMBP.

2.1 Synchronous Programming

The field of synchronous programming [35] offers a class of languages developed for writing control programs for embedded reactive systems. Esterel [6], Lustre [34] and Signal [31] are examples of synchronous programming languages that have been employed in industrial applications. The widely used StateCharts graphical specification formalism [37] shares key aspects of the synchronous programming model. In the design of TMBP, certain key ideas from the synchronous programming domain have been leveraged. This section highlights the similarities and fundamental differences between the textual and graphical TMBP languages (RMPL, THCA and TCCA), and one widely-used synchronous language, Esterel.

First, similarities between Esterel programming and TMBP are considered. Both

Esterel and RMPL/THCA include standard constructs for expressing reactive system behavior, such as conditional branching, iteration, parallel composition, sequential ordering and preemption. Berry [6] has convincingly argued that such features, as well as multiform time and determinacy, are necessary characteristics for reactive programming. RMPL is a synchronous language, and satisfies all these characteristics.[1] One major goal of synchronous programming is to provide "executable specifications," that is, to eliminate the gap between the specifications about which properties can be proven, and the programs that are supposed to implement these specifications. TMBP carries this idea one step further, by performing reasoning on executable specifications directly, in real time. Another important similarity is that both Esterel and RMPL compile to underlying automaton models with clean mathematical semantics. Both programming frameworks emphasize modularity in software design: Esterel uses the 'module' as its programming unit, while TMBP uses hierarchical, modular programs expressed as THCA and factored plant models expressed as TCCA (a modular composition of concurrent automata). Finally, like Esterel, RMPL is fully orthogonal, meaning that the constructs can be nested and combined arbitrarily.

Despite the similarities between Esterel and RMPL, there exist some fundamental differences in their corresponding philosophy. First, Esterel is a "signal-based" language, whereas RMPL/THCA is a "state-based" language. In Esterel, *signals* are logical objects received and emitted by a program, which are used to broadcast the occurrence of an event or communicate information throughout a system [7]. Rather than operating on signals, TMBP considers the notion of hidden system *states* to be the fundamental basis for execution. This different point of view is reflected in the way each language interacts with a physical system, as shown in Figure 1-2: Esterel programs interact with the program memory, sensors and control variables (but not directly with the plant state), by emitting and detecting signals. In contrast, timed model-based programs operate directly on system state, leveraging the executive's deductive controller to close the loop between state and the sensors and actuators.

[1] Reference [43] provides a mapping between constructs in Esterel and the corresponding RMPL forms.

This difference is attributable to the different application domains targeted by the two languages. For Esterel, which is primarily designed to provide coordination and synchronization between computational processes, signals and events are the most appropriate basic mechanisms. TMBP's task, which is to provide a framework for monitoring and control of complex dynamic plants, lends itself to thinking more naturally in terms of state evolution.

Another important difference is that the availability of instantaneous broadcasting and control transmission in Esterel makes it possible to write syntactically correct but semantically "non-sensical" programs, and programs whose behavioral semantics are non-deterministic given an input [6]. Such causality problems are avoided in RMPL/THCA, where constructs are conditioned on the current state of the physical plant, and act on the plant state in the next execution cycle.

2.2 Concurrent Constraint Programming

The concurrent constraint programming paradigm [72] evolved from a re-analysis of the ideas underlying synchronous programming, from the viewpoint of asynchronous computation. Like synchronous programs, concurrent constraint programs are declarative; they can be viewed as temporal logic formulas, with semantics based on solutions of equations. Concurrent constraint programming languages similarly define a set of basic combinators, from which programs are built compositionally. These basic combinators can be used to define numerous derived control constructs. Finally, like synchronous programs, concurrent constraint programs can be compiled into automata representations, which can be analyzed for guarantees of real-time properties.

Concurrent constraint programming languages, such as the Timed Concurrent Constraint Language (TCC) [33], replace the traditional embedded programming notion of an information store as a valuation of variables with the notion of a store as a set of constraints on program variables. These languages interact with the store by "telling" and "asking" constraints at consecutive time points (Figure 2-1). As is the

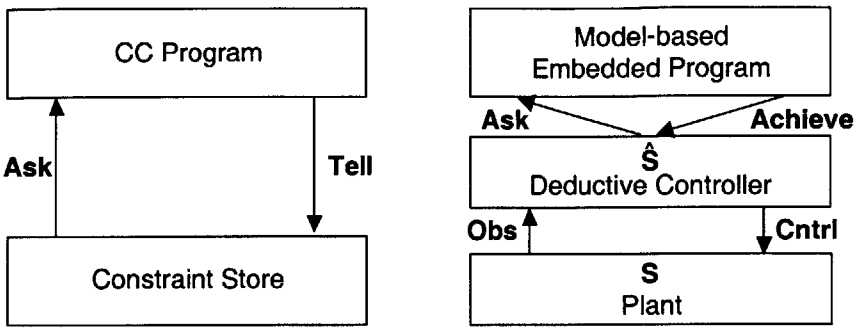

Figure 2-1: Model of interaction for concurrent constraint programming languages (left) and model-based programming languages (right).

case in synchronous programming, it is the programmer's responsibility to perform the mapping between intended state and the sensors and actuators. This mapping involves reasoning through a complex set of interactions under a range of possible failure situations. The complexity of the interactions and the number of possible scenarios make this an error-prone process. A model-based programming language leverages the benefits of both synchronous programming and concurrent constraint programming, with the key difference that it interacts directly with the plant state: state assertions (configuration goals) are specified as constraints on plant state variables that should be made true (*achieve* constraints, rather than *tell* constraints), and state condition checks are *ask* constraints. Further discussion of this distinction is found in Chapter 5.

TMBP and concurrent constraint programming share underlying principles, including the notions of constraint-based modeling and computation as deduction over systems of partial information [33]. Just as concurrent constraint programming offers a family of languages, each characterized by a choice of constraint system, TMBP defines a family of languages, each characterized by the choice of the underlying plant modeling formalism. TMBP extends concurrent constraint programming with a paradigm for exposing hidden states, a replacement of the constraint store with a deductive controller, and a unification of constraint-based and semi-Markov modeling.

2.3 Robotic Execution

State-of-the-art embedded executives are used to coordinate the run-time activity among various software modules within a control system. They must be able to respond quickly to events while bringing potentially large amounts of information (both system knowledge and real-time measurement data) to bear on their decisions [27]. In doing so, robust executives must manage interacting goal- and event-driven processes, while remaining reactive to contingencies. TMBP supports both goal- and event-driven execution, where goals and events are represented as changes to system states.

The robotic execution languages used to encode robust executives, like Reactive Action Packages (RAPs) [26] and Task Definition Language (TDL) [76], provide constructs for various task-level control capabilities. This section highlights the capabilities of a representative robotic executive and describes how the Timed Model-based Executive provides each of these capabilities.

The procedural executive EXEC was demonstrated onboard DS-1 as part of the Remote Agent Experiment [65]. Remote Agent's EXEC is written in a rich procedural language, the Execution Support Language (ESL) [27], which is an extension to multi-threaded Common Lisp. ESL provides language features that allow the encoding of execution knowledge into embedded autonomous agents. These features translate to the following key capabilities for EXEC, which are also provided by the Timed Model-based Executive: contingency handling, task management, goal achievement, time-keeping, logical database querying, and resource management. Each of these capabilities is discussed below.

1. **Contingency handling** – As discussed in Chapter 1, the Timed Model-based Executive is inherently fault-aware. The use of a deductive controller within the state-achievement loop enables the executive to detect and respond to failures "on-the-fly". Furthermore, through the combination of preemption constructs encoded in RMPL/THCA control programs and the hidden state diagnosis capability provided by the deductive controller, mechanisms for identifying and

recovering from failure contingencies and specifying cleanup procedures can be constructed.

2. **Task management** – Task management capabilities, such as spawning new concurrent tasks, aborting tasks, setting up task networks and defining guardian tasks (for task monitoring) are provided by parallel composition and preemption constructs in the timed control program. Task synchronization features, including signaling and waiting for particular events, are also provided by TMBP. By representing events as changes to states of the system or its environment, event-driven execution is straightforwardly accommodated.

3. **Goal achievement** – The mechanisms for specifying and commanding goal achievement methods in the TMBP framework are provided by the underlying deductive controller. The capacity for a Timed Model-based Executive to perform sequencing at the level of system state specifications, and to abstract away the details of how states are achieved, is considered a significant benefit.

4. **Time-keeping** – The introduction of clock variables, clock initializations, and time constraints in TMBP enables the Timed Model-based Executive to provide capabilities for timeout definitions and relative-time event scheduling. As discussed in Chapter 1, time is considered both at the level of the control specification and at the level of the plant behavior specification. This allows for execution conditioned on time as well as system state, and for reasoning about timed (semi-Markov) plant behavior.

5. **Logical database querying** – The need to explicitly maintain, query and reason about a distinct logical database is made obsolete by the presence of the deductive controller. This deductive controller maintains the latest system state knowledge and provides the control sequencer with all the state information it requires.

6. **Resource management** – The Timed Model-based Executive provides capabilities for basic resource management, using preemptive constructs in the

control program to trigger when a certain property, expressed in terms of state constraints, is no longer entailed. The current incarnation of RMPL/THCA does not provide advanced property locking mechanisms, to handle resource interactions between concurrent tasks. Incorporation of more advanced resource management features is a focus of current work.

In summary, the Timed Model-based Executive offers most of the goal-directed tasking and monitoring capabilities of robotic execution languages. A key difference is that TMBP fully covers synchronous programming, hence moving towards a unification of a goal-directed AI executive with its underlying real-time language. Furthermore, the incorporation of a deductive controller into the execution loop enables control sequencing to be performed at the level of system state specifications, and to abstract away the details of how states are achieved. This hidden state abstraction provides a powerful mechanism for goal- and event-driven execution.

2.4 Model-based Programming and Execution

In general terms, a model-based executive is defined as a reactive configuration management software module that uses a declarative specification of system behavior (plant model) to track system state and compute desired sequences of control actions. Model-based execution encompasses research in the fields of synchronous and concurrent constraint programming, robotic execution, model-based reasoning, diagnosis and reactive planning.

The first-generation model-based executive is the Livingstone system [83]; Livingstone is composed of an estimation component called Mode Identification (MI) and a control component called Mode Reconfiguration (MR). The MI capability builds off earlier work in model-based diagnosis (the GDE [19] and Sherlock [20] diagnostic systems) by including probabilistic transitions between modes of components in the plant model. This extension improves the diagnostic discrimination by allowing likely state estimates to be tracked through commanded reconfigurations of the system. To track the state of the plant, the Livingstone executive uses Sherlock's

any-time conflict-directed A* algorithm [20, 91] to efficiently search over the set of possible transitions in the model, in combination with a satisfiability engine based on an incremental truth maintenance system [59]. Livingstone employs these same algorithms and models in its MR capability, which computes a set of optimal control actions required to achieve a specified goal. Livingstone's MR has been upgraded to include the Burton reactive planner [85], which enables MR to reason through complex paths of interaction through the system, and generate multiple-action sequences that achieve the specified goals.

The core Livingstone MI and MR algorithms were reimplemented and improved by Kurien [50, 51]; the resulting model-based executive is named Livingstone2. Livingstone2 extends the MI capability, allowing it to incrementally generate, rather than revise, an approximate belief state by abstracting and summarizing segments of the likely plant state trajectories. This enables a system to maintain a partial belief state as long as it remains consistent with observations, and to revisit past assumptions about the state evolution when certain state estimates are ruled out. The MR capability is extended to provide safe planning in the presence of uncertainty or ambiguity in the current state estimate. It adopts an any-time conformant planning approach: given a set of possible initial states and a goal configuration, it chooses one of the initial states, finds a plan that achieves the goal for that initial state, and then incrementally tries to extend this plan into a new plan that is conformant for additional initial states.

Other extensions to the Livingstone algorithms were made by Ragno [67], van Eepoel [79], and Chung [16]. Ragno [67] first extended Livingstone by incorporating a complete DPLL-based satisfiability engine [17], replacing the original implementation's limited unit propagation-based satisfiability engine. He also demonstrated substantial performance improvement by replacing the conflict-directed A* algorithm (a weak coupling of A* search and satisfiability checking) with *clause-directed A**, an approach in which the search and satisfiability are more tightly coupled. Other work has focused on further improving the core Livingstone algorithm performance by "pre-compiling" its computationally expensive on-line model-based reasoning opera-

tions into a set of model-derived rules, which can be used to diagnose or command the spacecraft in time that is linear in the number of rules. A compiled version of mode estimation that implements approximate belief state update (as opposed to Livingstone's limited single-trajectory tracking capability) was developed by van Eepoel [79]. Chung [16] developed a compiled mode reconfiguration capability that extends the Burton reactive planning approach by leveraging transition-based decomposition and a compact symbolic encoding as Ordered Binary Decision Diagrams.

The next-generation model-based executive, called Titan [90], combined a deductive controller (evolved from Livingstone) with a procedural state-based control sequencing module, and formally introduced the model-based programming paradigm that the current TMBP work is based upon. By coupling deductive reasoning capabilities with the task control capabilities of an advanced procedural sequencer, the resulting executive demonstrates greater flexibility, fault-awareness and robustness than traditional executives.

A similar type of combined procedural/deductive executive was previously demonstrated as part of the Remote Agent Experiment (RAX), which flew on the DS-1 spacecraft in 1998 [60, 64]. In that case, the executive consisted of an integration of the EXEC system [65], with the Livingstone deductive executive [83]. This demonstration proved that the combination of knowledge encoded in procedures and in declarative models yields a rich modeling framework suitable for the control of spacecraft systems. However, the two components of the executive required a distinct knowledge representation, expressed using very different modeling languages. While such heterogeneous representations have a number of benefits, including the ability for different software components to reason at different levels of abstraction, they also present several difficulties. Most significant are the possibility for models to diverge and the need to duplicate knowledge representation efforts. One conclusion drawn from the Remote Agent design effort was the desire to head towards a unified representation of the spacecraft, while maintaining the ability to accommodate the complexities of the spacecraft domain and the capacity for knowledge abstraction [4].

The Titan model-based executive improves upon the design of the Remote Agent's

procedural/deductive executive, by embracing a cleaner separation of the roles of the procedural and deductive parts of the executive. The two modules require truly complementary knowledge bases, with the control programs representing state-based control specifications, and the plant models capturing system behavior within each state. This leads to less duplication of knowledge between the modules, which was a drawback of the Remote Agent's procedural/deductive executive. This duplication of knowledge resulted from Remote Agent's use of the Livingstone MR capability as a "recovery expert": MR suggested recoveries to EXEC rather than actually issue the commands it generated. This required EXEC's control programs to cover all the possible commands MR could generate. In contrast, the only common information shared by Titan's control programs and plant models is the set of system state variables.

The Timed Model-based Executive described in this thesis builds off the Titan executive, by folding time into both the control programs and plant models. The ability to express timed control programs allows the Timed Model-based Executive's control sequencer, described in Chapter 5, to accommodate time-critical sequences, like entry, descent and landing. The design of the Timed Model-based Executive's deductive controller, detailed in Chapter 6, extends Titan's algorithms for state inference and optimal system reconfiguration, to allow reasoning about timed plant models, capturing system behaviors that are characterized as semi-Markov.

Other work in model-based programming has also folded time into the computational model. The Kirk model-based executive [48] combines the flexibility of embedded programming and reactive execution languages, and the deliberative reasoning power of temporal planners. As introduced in Chapter 1, the Timed Model-based Executive focuses on reactively executing a timed control program expressed in terms of hidden state, and reasoning about semi-Markov behavior at the level of plant components. Kirk, on the other hand, operates at a higher level of abstraction: it provides capabilities for reasoning about activity-level contingencies, scheduling, and planning cooperative paths [88].

Kirk uses a non-deterministic control program representing a form of contingent

plan graph that encodes alternative activities with associated symbolic constraints and time bounds. Kirk's control programs differ from the timed control programs considered in this thesis, in that they allow for non-deterministic choice between activities and they do not operate directly on hidden plant state. Furthermore, Kirk control programs adopt a non-deterministic model of time, where time bounds are specified on activities, capturing the lower and upper limits on activity duration. In this sense, they resemble timed plant models in TMBP, except that they do not specify probabilistic time constraints.

Kirk reasons through its control program, choosing a set of activities that form a consistent (i.e., satisfies all symbolic constraints) and schedulable (i.e., satisfies all time constraints) plan. Kirk then executes its plan using a robust execution algorithm, described in [78], which adapts to execution uncertainties through fast online scheduling. Ongoing work in model-based programming is underway to extend the Kirk paradigm to distributed planning and execution problems [81], and to unify Kirk's capabilities for scheduling and contingency planning with the Timed Model-based Executive's capabilities for reactively operating on hidden states of semi-Markov plants.

2.5 Formal Modeling of Real-Time Systems

System specification languages for real-time system modeling, such as Timed Automata [3] and Timed Transition Systems [39], are characterized by their clean semantics and amenability to verification via formal tools. Widely used model-checking tools, such as SPIN [41], KRONOS [10], UPPAAL [55] and SMV [12], are used to verify whether a system described using a formal system specification language satisfies certain state properties, including reachability, safety, and progress.

The incorporation of time into the formal semantics of TMBP borrows various ideas from the semantic descriptions of these system specification languages. For example, both paradigms define a complex system as a composition of concurrently-operating automata. Furthermore, like most existing semantic models that describe

timing-based systems, TMBP adopts an *interleaving* model of computation, where an execution is represented as an alternating sequence of instantaneous "discrete" events and "continuous" phases. Consequently, legal executions of a timed model-based program are defined as discretized timed state sequences (see Chapter 4) satisfying all timing constraints on transitions; this definition is similar to the definition of *computations* of a Timed Transition System [39].

Key ideas from these formal system specification languages are also folded directly into the languages used to implement timed model-based programs, at both the control program and plant model levels. The THCA graphical language used for control programs defines clock variables, clock interpretations, clock initializations, and timing constraints, elements common to modeling formalisms like Timed Automata [3]. However, whereas THCA are intended to provide a framework for executable specification of embedded control programs that run "in the loop" as part of a system's real-time control system, formal system specification languages are intended to provide a framework for off-line formal verification and model checking. This fundamental difference in intent leads to some key differences in the language, such as THCA's adoption of a hierarchical computational model. Another formal specification language, Timed Statecharts [47], also adopts a hierarchical structure, but THCA are distinguished by their use of state-based configuration goals as a mechanism for goal-driven execution. A more detailed discussion of the comparison between formal specification languages and THCA is found in Chapter 5.

At the level of the plant models, the TCCA formalism has similarities with a variant of Timed Automata, called Probabilistic Timed Automata [52, 53]. Both formalisms describe system behavior through a set of concurrently-operating automata, where probabilistic transitions between states are conditioned on event labels and clock constraints. There are, however, some key differences between the two representations due to the constraint-based encoding of TCCA and its use for modeling semi-Markov plant behaviors. For instance, TCCA use general propositional logic constraints on plant variables to specify behavior in each component mode, whereas Probabilistic Timed Automata simply associate with each node a set of atomic propo-

sitions that are true in that node. A more detailed discussion of the differences between the TCCA and Probabilistic Timed Automata formalisms is provided in Chapter 6.

Recent work in formal verification has studied the use of model-checking techniques to validate plant models for untimed model-based executives [61, 63]. Research in applying these methods to timed plant models and timed control programs is identified as an area for future work (see Chapter 8).

Finally, it is interesting to note that recent research has seen formal models and model-checkers used for diagnosis [53, 54]. Chapter 6 includes further discussion of this work in comparison with the mode estimation approach used in the Timed Model-based Executive.

2.6 Mission Data System

The Mission Data System (MDS) is an embedded software architecture, currently under development at NASA JPL. Its overarching goal is to provide a multi-mission information and control architecture for the next generation of robotic exploration spacecraft, that will be used in all aspects of a mission: from development and testing to flight and ground operations. In the process of achieving this ambitious goal, the MDS team has rethought the traditional mission software lifecycle, and has adopted a vision that acknowledges and leverages the intimate coupling between software and systems engineering: "Software is part of and contributes substantially to a new systems engineering approach that seamlessly spans the entire project breadth and life cycle." [22]

Though its scope and objective are much broader, MDS shares numerous architectural themes with TMBP:

1. **Take an Architectural Approach**

   In traditional approaches to designing mission software, code is "compartmentalized" in various ways, e.g. flight vs. ground vs. test, and by subsystem (power, thermal, navigation, etc...). As a result, each subsystem's software en-

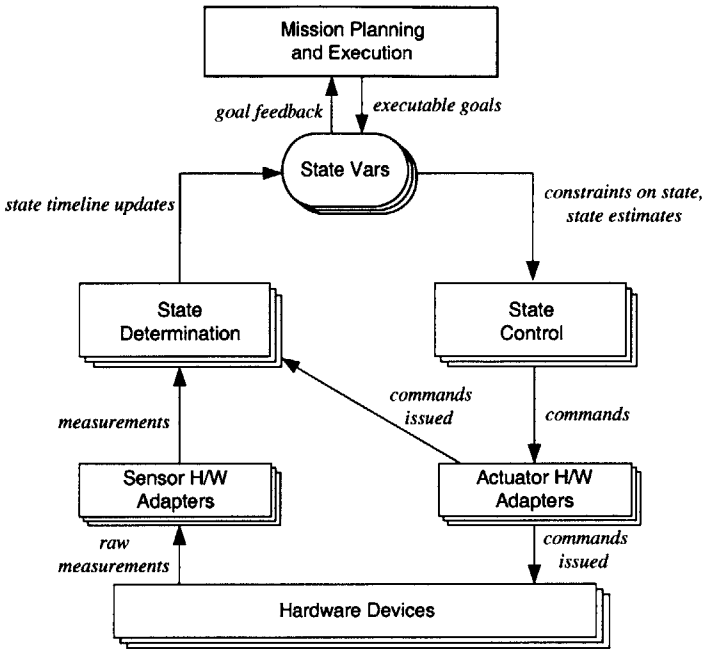

Figure 2-2: Simplified MDS architecture diagram.

gineering and programming teams tend to apply customized solutions to problems in their subsystem, leading to minimal amounts of software reuse across the different subsystems. Furthermore, in addition to the problem of inefficiency, this approach leads to increased interface complexity between subsystems, as many important mission considerations (such as onboard resource limitations) introduce coupling across subsystem boundaries [23]. Getting the interfaces right generally requires many iterations on the part of multiple subsystem software teams.

In contrast, MDS adopts an architectural approach, meaning that it strives to identify common problems and provide common solutions, in the form of shared core architecture elements, such as estimators, controllers and planners. Subsystems are constructed from architectural elements, instead of the other way around. This decreases the amount of redundant (and potentially conflicting) code written, and promotes consistency through common resource coordination services provided by the architecture. A simplified interpretation of the MDS architecture is shown in Figure 2-2.

TMBP adopts a similar philosophy. The three main architectural modules (control sequencer, mode estimation and mode reconfiguration) are used for system-wide control. Component- and subsystem-specific behavior information is embodied in the plant models, with couplings specified through interconnection constraints; the executive's deductive controller has the responsibility of managing these couplings in a system-wide way, rather than locally, based on feedback from the spacecraft sensors and goals from the control sequencer. Similarly, subsystem-specific control information is captured in control programs, for system-level management by the control sequencer.

2. **Consider State and Models to be Central to the Architecture**

   In traditional approaches to embedded software, and spacecraft flight software in particular, programs are written such that they prescribe the desired state evolution implicitly, through low-level commands to actuators and references to sensors. Their implicit consideration of state makes these programs hard to encode and verify.

   Like TMBP, MDS is a *state-based* architecture, where state is defined as the momentary condition of a dynamic system. State is accessible in a uniform way through state variables, instead of through local variables in the control program. Both paradigms emphasize the separation of application-specific knowledge, in the form of models that describe how state evolves, from reusable general-purpose code that operate on the models to track and control state. The novelty in this approach is that models are used explicitly, rather than being "hidden" in the details of the control program, as in traditional flight code. This leads to easier portability from mission to mission, as only the models need to be updated with domain-specific knowledge.

   In keeping with its broader scope, MDS adopts a more general representation of state than TMBP. Unlike TMBP, which currently operates only on discrete state models, MDS considers both continuous and discrete state. Furthermore, state evolution in MDS is described on state timelines, which provide a record

of current estimates, past estimates, and future predictions. The concept of state timelines, while not inconsistent with TMBP, has not been implemented within the paradigm. The current implementation of the Timed Model-based Executive focuses on using current state estimates to direct the execution of control programs.

The concept of models is also more general in MDS than in TMBP. MDS models can be tables, functions, rules, state machines, etc... The plant models used in the current TMBP implementation are encoded as a specific form of factored POSMDP, described in Chapter 6. However, it should be noted that the TMBP paradigm conceptually defines a broader family of languages, each distinguished by the choice of semantic model for the plant. This allows for different implementations of Timed Model-based Executives that operate on behavior models represented in different forms.

3. **Enable Goal-Directed Closed-Loop Operation**

   Traditional spacecraft control programs essentially consist of unconditional, time-tagged sequences of commands to be issued, generally in an open-loop manner, based on implicit assumptions of state. These sequences must be painstakingly designed by systems engineers on the ground, who must reason about the complex end-to-end spacecraft system to successfully predict the resulting spacecraft behavior and ensure it follows the desired trajectory. Beyond the "low-level" controllers specified for particular subsystems, such as the attitude control or thermal subsystems, onboard system-level monitoring and response capabilities are generally limited, involving ground-controllers in the loop and a fault-monitoring/safing system that operates in parallel with the nominal flight software.

   This operational model can lead to lengthy control specifications, due to the command-level detail of sequences. Furthermore, it can lead to brittle behavior, due to its very limited branching ability and the lack of onboard consideration of the *intent* of the sequence. MDS and TMBP both address these problems:

instead of issuing low-level open-loop commands, they issue *goals* that indicate intent in the form of desired state. Goals are easier to specify than the actions needed to achieve them, and result in more compact specifications of desired behavior. Furthermore, goal-directed operation goes hand-in-hand with closed-loop control, because goals can be thought of as set points for onboard controllers, which are then given the latitude to decide how best to achieve the goals. The latter role is filled by the deductive controller in a Timed Model-based Executive. For MDS, these onboard controllers are known as *goal achievers.* It should be noted that TMBP's definition of a goal, as a constraint on state variables that must be satisfied, is not as general as MDS' definition of goal, as a prioritized constraint on the value of a state variable *over a specified time interval.* This distinction is consistent with the emphasis on state timelines in MDS.

4. **Separate State Determination from State Control**

   Unlike traditional approaches to embedded software, in which control logic is intermingled with state determination logic, MDS and TMBP advocate making a clear separation between these two key functions, which are coupled solely through state variables. Taking this approach allows state knowledge to be updated in a unified, consistent way, for use by any control function in the architecture. For instance, in the Timed Model-based Executive, state estimates from the mode estimation module are used both by the control sequencer and the mode reconfiguration module. In MDS, multiple controller modules might need to access the same state variable. Keeping the state determination and state control functions separate ensures that all active controllers use a consistent estimate of state. Furthermore, this type of increased modularity simplifies module-level testing of various state determination or control algorithms, and allows for minimally invasive upgrades to individual state determination and control modules.

5. **Provide Integrated Fault Protection**

The goal-directed nature of the MDS and TMBP paradigms leads to intrinsic *fault-awareness* in the system; that is, fault detection, diagnosis, and recovery are an integral part of the design of the architecture. This is in contrast with traditional flight software, where fault protection is provided as an "add-on" capability, running in parallel with the nominal sequence execution code. Intrinsic fault-awareness is enabled by providing the executive with knowledge of the operational intent (in the form of state goals), and the ability to derive appropriate actions by reasoning about the state of the system, instead of by edict [69]. In both paradigms, fault states are included in the behavior models and are treated just like any other nominal state. Fault detection and diagnosis are thus performed by the state determination (mode estimation) modules, whenever the observed system behavior is different from the behavior dictated by the current nominal mode estimate. This diagnosis process is performed in the same loop as the nominal state tracking process. Similarly, in the case of a diagnosed failure, recovery is also handled by the same goal achieving controller in charge of performing the nominal system control actions (assuming such a recovery is possible, of course; if it isn't, the goal achiever's job is to signal failure of the goal to the module that issued the goal).

Another key element of robust operations is the consideration of knowledge uncertainty. In the MDS framework, state knowledge uncertainty is tracked in an explicit way, within the state variables. In TMBP, state uncertainty is built into the plant models in the form of probabilistic transitions, and is considered in the process of mode estimation. However, more sophisticated capabilities, such as conditional execution based on the level of confidence in the state estimate, are not currently provided by the Timed Model-based Executive (except to the extent by which such reasoning can be captured within the plant model).[2]

[2]For example, consider the case where a given state is determined by mode estimation to be the most likely, but that its associated probability is only 60%. It would seem reasonable, in certain

MDS also provides the ability to issue goals on knowledge quality/certainty, a capability that is not presently built into the Timed Model-based Executive. MDS provides a good model for the TMBP paradigm, with respect to improving its mechanisms for handling state uncertainty.

To summarize, both the MDS and TMBP paradigms advocate that *system state* and *models* form the foundation for monitoring and control. As stated above, MDS' goal is to provide a unified architecture and a set of component frameworks to accommodate appropriate technologies in support of a broad spectrum of space missions. Model-based execution technology bridges the gap between system-level planning (a capability provided by MDS' Mission Planning and Execution module) and real-time subsystem commanding (provided by goal achievers in MDS). Consequently, an effort is currently underway to infuse various elements of the Timed Model-based Executive into MDS, including the control sequencer, mode estimation and mode reconfiguration. Each of these component technologies has been identified as complementary to baseline MDS functions. The control sequencer provides a capability for execution-time goal elaboration, closing a more reactive loop than the MPE goal failure and replanning cycle. The mode estimation and mode reconfiguration engines can play an important role at the goal achiever level, providing a system-wide deductive estimation and control capability not already provided by MDS core framework elements. For example, Figure 2-3 illustrates a proposed integration of model-based mode estimation into MDS' state determination framework. This infusion effort is being carried out in the context of the first "customer" mission for MDS, Mars Science Laboratory, a rover mission scheduled for launch in 2009.

situations, to avoid pursuing the nominal execution sequence conditioned on that state, in favor of taking another course of action that is perhaps more conservative, or might provide more confidence in the state estimate. The current Timed Model-based Executive implementation does not allow this type of behavior.

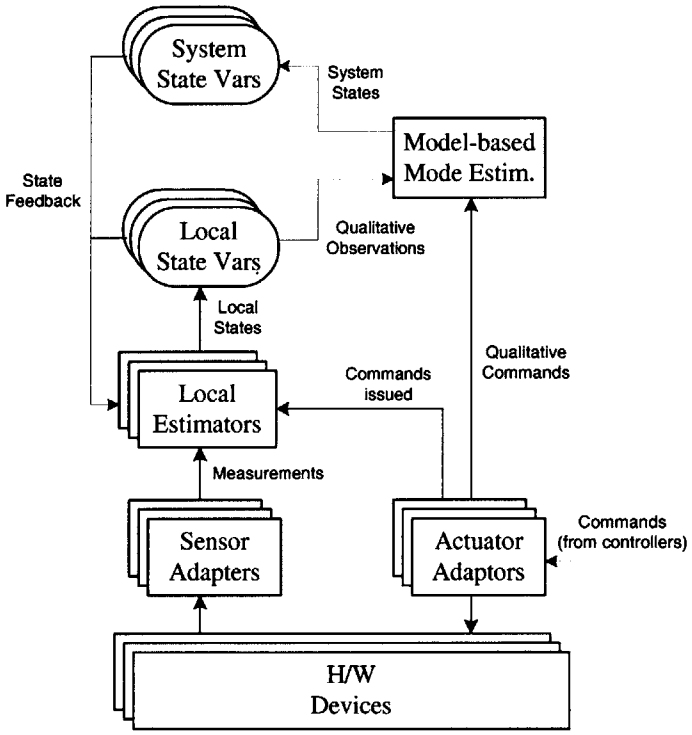

Figure 2-3: Integrating model-based mode estimation into the MDS state determination framework.

Chapter 3

Timed Model-based Programming Example

As explained in Chapter 1, the execution of complex critical spacecraft sequences depends on timing conditions, as well as knowledge of spacecraft state. As an example, consider the entry sequence for a Mars lander spacecraft, such as the Mars Polar Lander [92] (Figure 3-1).

At the end of the cruise phase of its mission, as the spacecraft approaches Mars, it turns on and heats up its descent engine, putting it into standby mode. Four and a half hours later, as the spacecraft nears its entry point into the Martian atmosphere, it switches from Earth-relative navigation, using a combination of a star tracker and an inertial measurement unit (IMU), to inertial navigation using only the IMU. This navigation mode switch is necessary because, once atmospheric entry is initiated, the spacecraft will no longer be able to perform the reorientations necessary to track the reference stars. Four minutes after switching its navigation mode, the spacecraft prepares for atmospheric entry by rotating to its entry orientation. Once the entry orientation has been achieved, the lander stage of the spacecraft separates from the cruise stage and proceeds toward entry into the Martian atmosphere (all the while holding its attitude at the entry orientation). When atmospheric entry is initiated (as determined by a change in the spacecraft's acceleration due to atmospheric drag), the entry sequence ends and the spacecraft proceeds to the descent and landing phases of

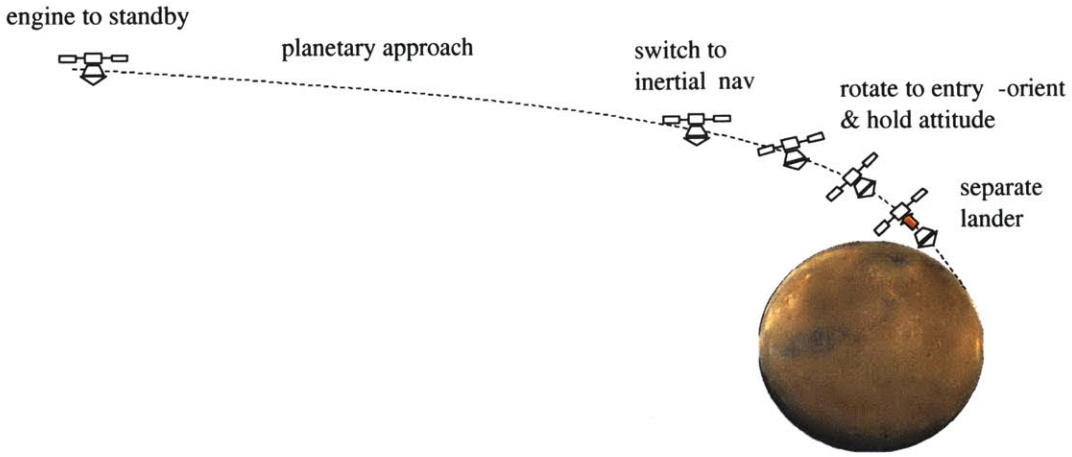

Figure 3-1: Mars entry sequence.

the mission. The full entry, descent and landing sequence is described in Chapter 7, in the context of a demonstration of the implemented Timed Model-based Executive.

In the above sequence, execution is conditioned on time conditions. For example, consider the four and a half hour delay between putting the descent engine into standby mode and switching to inertial navigation. Though it might seem more desirable to condition the sequence execution on a state of the spacecraft, in this case the relative distance of the spacecraft from the entry point, this is not an option: the spacecraft does not have access to any observations that would allow it to measure this relative distance. For this reason, the sequence must include a hard-coded delay, the length of which has been conservatively (but fairly accurately) estimated from computations performed by ground operations personnel based on the cruise trajectory of the spacecraft. The onboard executive must therefore have the ability to initiate clocks that it can reference, in order to check for satisfaction of timing conditions.

The executive must also consider time as it reasons about its hardware behavior. For example, the process of putting the descent engine into standby mode involves turning on and heating up the engine. Based on engineering specifications of the engine design, the spacecraft should know that this heating-up operation nominally takes from 30 to 60 seconds to complete. Based on available engine temperature

measurements, the onboard executive should then be able to confirm achievement of the standby state. If the sensors indicate that the engine has not reached its nominal standby temperature within the specified nominal time window, the executive should deduce that the engine has somehow malfunctioned, and that it should attempt to either repair the fault or work around it by leveraging any onboard redundancy. To be able to perform such time-based reasoning, the executive needs to have a mechanism for keeping track of the amount of time each modeled spacecraft component spends in each state.

It is also important to note that this sequence specification is expressed in terms of states of the spacecraft. Many of the states in the sequence are "hidden"; that is, they are not directly observable, but instead must be deduced indirectly based on one or more sensor observations and knowledge of how these observations relate to the state of interest. For instance, the state of an engine must generally be deduced based on various measurements of electrical power, temperature, and acceleration.

Similarly, hidden states are not necessarily directly controllable, but instead must be commanded indirectly, sometimes through a complex communication path conditioned on states of multiple components. Considering the same descent engine, for example, engine commands from the flight computer must pass through a propulsion drive electronics module, which must be powered on for the commands to reach the engine.

This type of state-based specification is far simpler than a control program that must turn on heaters and valve drivers, open individual valves in the propulsion subsystem, and interpret readings from the various sensors in the system. Having engineers specify desired spacecraft behavior in terms of more abstract hidden states makes the task of writing the control program much easier and avoids the error-prone process of reasoning through low-level system interactions. In addition, it gives the program's execution kernel the latitude to respond to novel failures as they arise, as described above for the engine example. This is essential for achieving high levels of robustness.

The remainder of this chapter informally introduces the timed model-based pro-

```
1   EntrySequence() :: {
2     engine = standby;
3     t1 = 0;
4     when (t1 ≥ 270.0) donext {
5       nav = inertial;
6       t2 = 0;
7       when (t2 ≥ 4.0) donext {
8         do {
9           always (att = entry-orient),
10          when (att = entry-orient) donext (lander = separated)
11        } watching (entry = initiated)
12      }
13    }
14  }
```

Figure 3-2: RMPL timed control program for the Mars entry sequence.

gram corresponding to the Mars entry sequence described above. It begins by specifying the two components of the timed model-based program: the timed control program and the timed plant model. It then describes the execution of the program under nominal and failure situations.

3.1 Timed Control Program

The above discussion illustrates that execution of a critical sequence, like Mars entry, depends on timing conditions as well as knowledge of spacecraft state. The RMPL control program shown in Figure 3-2 provides a textual encoding of the informal specification given previously as a set of state trajectories. The specific RMPL constructs used in the program are introduced in Chapter 5.

The timed control program begins by placing the engine in the standby state ("*engine = standby*"), and then notes the current time by initializing a clock variable ("*t1 = 0*"). This sequence of actions is performed by lines 2–3, where a semi-colon at the end of a line denotes sequential composition. The program then waits until 270 minutes have elapsed (line 4) before switching the navigation to inertial mode ("*nav = inertial*" on line 5). After initializing a second clock and waiting 4 minutes (lines 6–7), the program performs two activities concurrently (the comma at the end of line 9 indicates parallel composition): it commands the spacecraft to rotate to its

entry orientation ("*att = entry-orient*"), and then hold this orientation indefinitely (line 9); concurrently, the program waits for the entry orientation to be achieved, then commands the lander spacecraft to separate from the cruise stage ("*lander = separated*" on line 10). These concurrent activities are preempted as soon as the spacecraft is determined to have begun atmospheric entry (line 11), by watching for entry to be initiated ("*entry = initiated*") and terminating as soon as that condition is observed.

This timed control program highlights several of the features of TMBP, discussed in Chapter 1. First, the program is stated in terms of state assignments, such as "*engine = standby*", clock initializations (such as "*t1 = 0*"), and clock constraints (such as "*t1 < 4.0*"). Second, state assignments appear both as goal assertions and execution conditions. For example, in line 9, "*att = entry-orient*" appears in a goal assertion, while in line 10, it appears in an execution condition. Third, these state assignments are generally not directly observable or controllable; they must be deduced from the spacecraft's sensors (e.g., accelerometers, gyros, and temperature sensors) and controlled indirectly through flight computer commands acting on the spacecraft actuators (e.g., attitude control thrusters, reaction wheels, and heaters). Fourth, time constraints are used to capture unmodeled information about the spacecraft or its environment. For example, the 270-minute delay reflects the change in relative position of the spacecraft with respect to the Mars atmospheric interface, based on the cruise trajectory computations performed by ground controllers. Finally, by referring to hidden states directly, the RMPL program is far simpler than a corresponding program that operates on sensed and controlled variables. The added complexity of the latter program is due to the need to fuse sensor information and generate command sequences under a large space of possible operation and fault scenarios.

3.2 Timed Plant Model

The timed plant model is used by a Timed Model-based Executive to map queried and asserted states in the timed control program to sensed variables and control

sequences, respectively, in the physical plant. The timed plant model is built from a set of component models. Each component is represented by a set of component modes, a set of constraints defining the behavior within each mode, and a set of timed probabilistic transitions between modes. Chapter 4 describes the semantics of timed plant models in terms of factored Partially Observable Semi-Markov Decision Processes.

For the Mars entry example, the spacecraft is represented by supplying state models for "components" (not necessarily physical components), such as those depicted graphically in Figure 3-3.[1] Nominally, an engine can be in one of four modes: *off*, *heating*, *standby*, or *firing*. The behavior within each of these modes is described by a set of constraints on plant variables with finite domains, namely *thrust*, *power*, and *temperature*. In Figure 3-3, these constraints are specified in boxes next to their respective modes. These constraints are expressed as logical relationships between plant variable assignments. The engine also has a *failed* mode, capturing any off-nominal behavior. The possibility that the engine may fail in an unexpected way is always entertained, by specifying no constraints for the engine's behavior in the *failed* mode. This approach, called *constraint suspension* [18], is common to most model-based diagnostic approaches [20, 83].

Models include timed, commanded and uncommanded transitions, all of which are probabilistic. For example, the engine has uncommanded transitions from *off*, *heating*, *standby*, and *firing* to *failed*. These transitions each have a 0.1% probability. Such uncommanded transitions are always enabled, capturing the possibility of an unexpected fault occurring at any time. Transitions between nominal modes are conditioned on commands and/or time constraints. Time constraints are expressed in terms of clock variables, which capture the amount of time spent in a given mode. For example, the engine's transition from *off* mode to *heating* mode is conditioned on the command "*cmd* = *standby*." The transition from *heating* to *standby* mode is

[1] It should be noted that the models described here provide a fairly abstract representation of component behavior. However, timed plant models can also be written at lower levels of abstraction; for example, a spacecraft engine could be modeled as a composition of more detailed component models of valves, thrusters, heaters, etc...

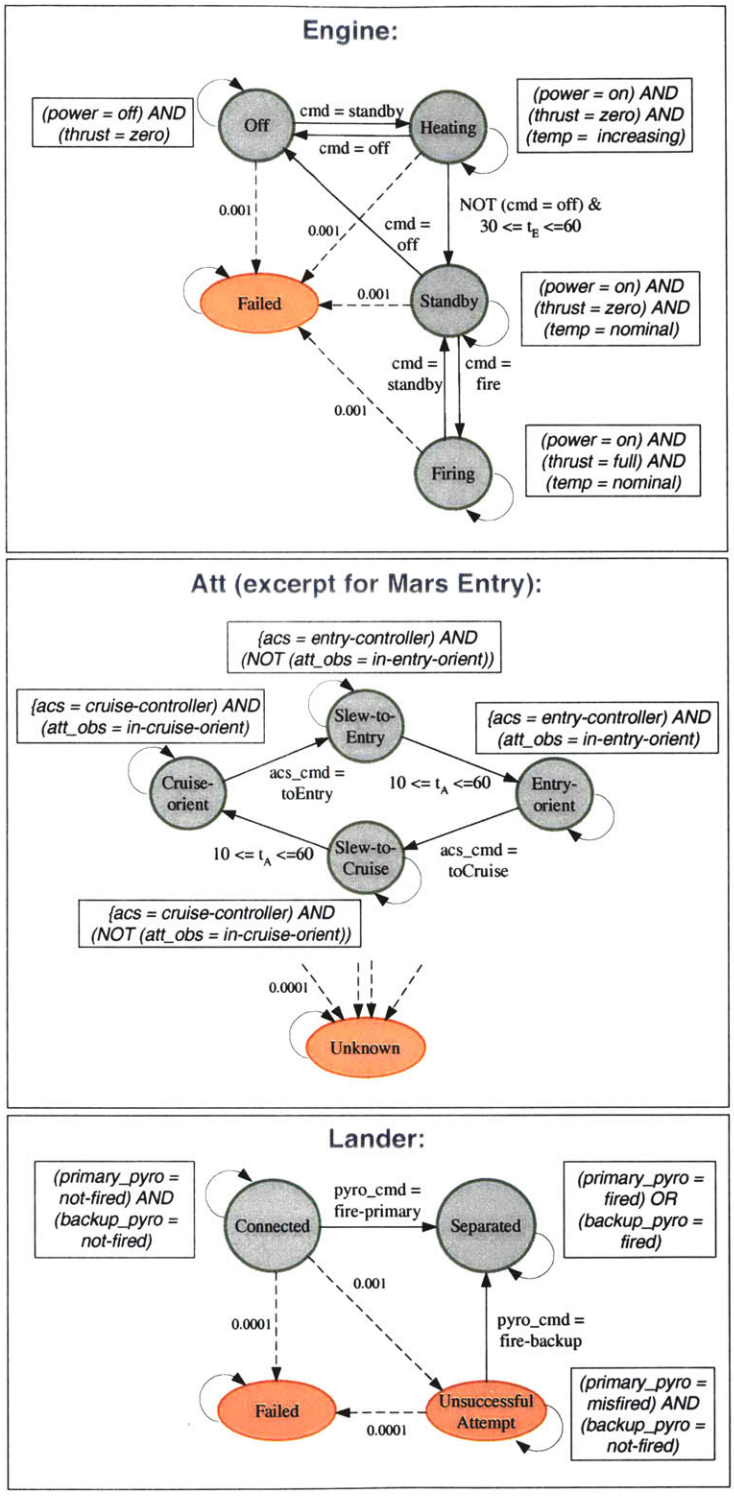

Figure 3-3: Example state transition models for a simplified Mars lander spacecraft. Nominal modes are represented as circles, and fault modes are represented as ovals. The probabilities on nominal transitions are omitted for clarity.

conditioned on the time constraint "$30 \leq t_E \leq 60$," as well as the non-issuance of the command "$cmd = off$," which would result in the transition back to *off* mode. In this example, all commanded nominal transitions occur with probability 99.9%, immediately upon assertion of their triggering commands. The timed transition is associated with a uniform probability density function over the time interval [30, 60] that integrates to 99.9%, representing the fact that the timed transition from *heating* to *standby* can occur at any time from 30 to 60 seconds after the engine has entered the *heating* mode. This implies that the timed plant model correspond to a partial specification of system behavior, which can have many consistent executions, even in the case of nominal behavior. This is an important distinction of timed plant models, as compared to the untimed plant models used in model-based programming [90].

The full plant model for the simplified spacecraft used in the Mars entry example includes component models for the *engine*, *nav*, *att*, *entry* and *lander* states. The full set of models is provided in Appendix C. Further detail on plant modeling is deferred to Chapter 6.

3.3 Model-based Program Execution

Figure 3-4 shows the graphical representation of the timed control program for the entry sequence discussed in Section 3.1. This representation, called a *Timed Hierarchical Constraint Automaton*, consists of a set of locations (represented as circles and boxes in the figure) arranged in a hierarchy, in the spirit of StateCharts [37]. While they are marked, locations can assert configuration goals, corresponding to states that the plant must progress toward (e.g., $nav = inertial$ is asserted in the location labeled "5"). Locations can also assert clock initializations (e.g., location 3 initializes clock *t1* to zero). Transitions between locations (represented as arrows in the figure) can be conditioned on time and/or state constraints (e.g., the transition from location 4 to location 5 is conditioned on the time constraint $t1 \geq 270\ min$). These automata will be formally defined in Chapter 5 of the thesis. Here, general references are made to the figure, to provide an informal idea of how timed control programs are written

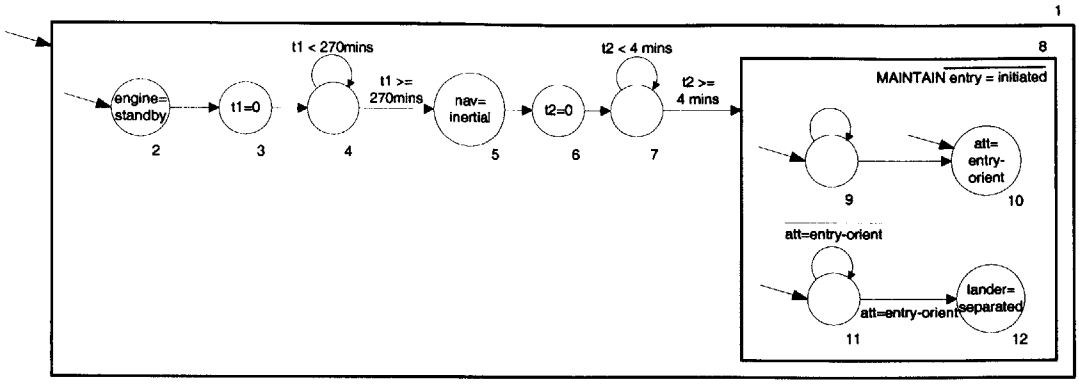

Figure 3-4: THCA representation of the Mars entry sequence.

and executed. In Chapter 5, this example will be revisited and its execution will be discussed in more detail.

Recall that the Timed Model-based Executive is composed of two modules, the control sequencer and the deductive controller. The control sequencer's role is to execute the timed control program by issuing appropriate configuration goals, based on the current state estimate and the current system time. The deductive controller's role is to generate the current state estimate based on observations from the plant and the current system time, and to issue commands that move the plant through a sequence of states that achieve the configuration goals.

The execution of this timed control program begins by asserting the configuration goal *engine = standby* (at the location labeled "2" in Figure 3-4), which the control sequencer issues to the deductive controller for achievement. To determine how to achieve this goal, the deductive controller considers the latest estimate of the state of the plant. Suppose the deductive controller determines from its sensor measurements and previous commands that the engine is off. The deductive controller deduces from the model that it should send a command to the plant that will lead the engine to standby mode. Based on the "*engine*" model in Figure 3-3, the deductive controller issues the command *cmd = standby*. Based on new sensor measurements confirming that the engine is indeed powered on, that its temperature is increasing and that the thrust is still zero, the deductive controller tracks the engine's transition into heating mode. Since the configuration goal *engine = standby* is not yet achieved, the

control sequencer continues to assert this goal. The deductive controller determines from the model that the standby mode will be reached eventually, because of the timed transition from the heating mode, without issuing any explicit command. At each execution cycle, the deductive controller considers the amount of time elapsed since the engine's transition into standby mode, and computes the likelihood that the timed transition is taken. It also checks the latest updates to the observables, to provide confirmation of the transition into standby mode. After 43 seconds, the engine temperature is observed to reach its nominal level and the deductive controller deduces that the transition into standby mode has been taken.

With the configuration goal now achieved, the sequencer then initializes a clock variable (at location 3) and waits for the clock to read 270 minutes (at location 4). At this point, the control sequencer asserts the goal *nav = inertial* (at location 5). The deductive controller determines that the goal can be achieved by simply issuing a command that triggers the desired change in navigation mode. After receiving confirmation that the goal has been achieved, the sequencer initializes a new clock (at location 6), and waits for 4 minutes to elapse (at location 7). At this point, it starts continuously asserting the configuration goal *att = entry-orient* (at location 10, which gets continually re-marked by the transition from location 9), which the deductive controller begins to achieve by commanding the appropriate mode switch for the attitude control system (*acs_cmd = toEntry*, see the "*att*" model in Figure 3-3). When the entry orientation is achieved (transition from location 11 to 12) after a 12-second slew, the sequencer proceeds by issuing the configuration goal *lander = separated* (location 12), all the while continuing to assert *att = entry-orient*, to maintain the spacecraft's attitude. This results in the deductive controller triggering the firing of the lander's pyro latches to separate it from the cruise stage. The sequencer continues to hold its entry orientation until the deductive controller indicates that entry has been initiated (*entry = initiated*), based on IMU sensor measurements indicating the onset of drag due to atmospheric entry, at which point execution of the Mars entry control program in Figure 3-4 terminates.

This describes a nominal (i.e. fault-free) execution of the entry sequence. How-

ever, the robustness provided by the TMBP approach is particularly emphasized in the case of off-nominal execution. One of the main strengths of TMBP is its fault-awareness, i.e. its seamless incorporation of fault diagnosis and recovery capabilities within the sense-decide-act loop. Consider a hypothetical situation where the primary latches connecting the lander to the cruise stage fail to release upon command (corresponding to the fault transition into the *unsuccessful-attempt* mode for the "*lander*" model in Figure 3-3). Since the timed control program for the entry sequence has specified the configuration goal *lander* = *separated* (at location 12), the control sequencer will continue to assert this goal to the deductive controller until it has been achieved, or until it has been determined that the goal state cannot possibly be achieved from the current estimated state. Presuming that the mission-critical pyro latch subsystem incorporates some redundancy, failure of the primary latches to fire open would result in the deductive controller reasoning through the "*lander*" model to deduce that it should fire the backup latch. Whereas current approaches to spacecraft fault protection would require explicit diagnosis and recovery actions to be built into the sequence, the Timed Model-based Executive can perform this recovery in a manner that is transparent to the control sequencer.

3.4 Summary

This chapter has provided an informal overview of TMBP by describing the timed model-based program for a simple lander spacecraft and a Mars entry sequence. Nominal and off-nominal executions of the program have been described, highlighting the interactions between the control sequencer and deductive controller modules of the Timed Model-based Executive.

The key insights to extract from this chapter are listed as follows:

- the execution of critical spacecraft sequences is conditioned on both time and state, as illustrated in the timed control program for Mars entry;
- the components that make up the physical plant can exhibit timed behaviors,

as illustrated by the engine's heating process, about which the executive must be able to reason;

- the Timed Model-based Executive is fault-aware, in that it has the ability to detect and manage faults, within its nominal goal achievement loop.

In the following chapter of the thesis, a formal description of the semantics of Timed Model-based Execution is provided. Subsequent chapters describe the implemented framework for specifying and executing timed model-based programs, and illustrate their execution in more detail.

Chapter 4

Timed Model-based Programming and Execution Semantics

As described in Section 2.2, the concurrent constraint programming paradigm [33] offers a family of languages sharing a common semantics, with each language characterized by a choice of constraint system. Similarly, TMBP defines a family of languages, each characterized by the choice of the underlying plant modeling formalism and the implementation of the associated deductive controller. This section presents a semantic model for the family of TMBP languages, and the Timed Model-based Executive that operates on a timed model-based program. The implementation of the executive, described in Chapters 5 and 6, provides a computationally tractable approximation to the abstract executive semantics presented in this chapter.

This chapter begins by describing the semantics for the two parts of the timed model-based program: the timed plant model and the timed control program. The semantics for the control sequencer and deductive controller modules of the Timed Model-based Executive are then presented. The chapter concludes with a semantic definition of the execution of a timed model-based program in terms of legal state evolutions of a physical plant. The execution semantics for a timed model-based program is one of the key contributions of this thesis.

4.1 Timed Plant Model

Previous work in model-based programming and execution used a factored Partially Observable Markov Decision Process (POMDP) model to describe a physical plant [90]. However, for timed plant models of the type introduced in Section 3.2, the POMDP model is inadequate. It does not provide the ability to represent timed state transitions, which are necessary to model system behaviors such as delayed reactions and gradual state evolutions. Rather, this work models the plant as a factored variant of a *Partially Observable Semi-Markov Decision Process (POSMDP)* [58]. The term *factored* is used here to indicate that the system model is composed of component models. Each component model, corresponding to a variant of a POSMDP, has an associated *state variable* and *clock variable*. The state variable stores the current *mode* of the component. The set of all current component mode assignments is referred to as a *plant state*. The clock variable stores the amount of time elapsed since the component transitioned into its current mode. A dense model of time is assumed, where the time domain is taken as the set of non-negative real numbers, $\Re^+$.

The POSMDP model used in this work has some notable differences from the "standard" POSMDP model described by Mahadevan [58]. The key distinction of the POSMDP model presented here lies in the assumption that decision points (referred to in the literature as *decision epochs*) are sufficiently frequent that the state does not change more than once between decision epochs, and a state change (if one occurs) can be assumed to happen at the instant of the following decision epoch. This differs from the POSMDP model presented in [58], in which decision epochs occur at random points of time determined by a probability distribution and in which the state of the system can change between decision epochs. This distinction is illustrated in Figure 4-1, which shows representative evolutions of state in the system over time for both POSMDP models, and the decision epochs at which observations of the behavior and decisions are made. The difference in the POSMDP model adopted here is due to TMBP's use of the model to capture low-level plant behavior, where a higher frequency of control interaction with the plant is necessary.

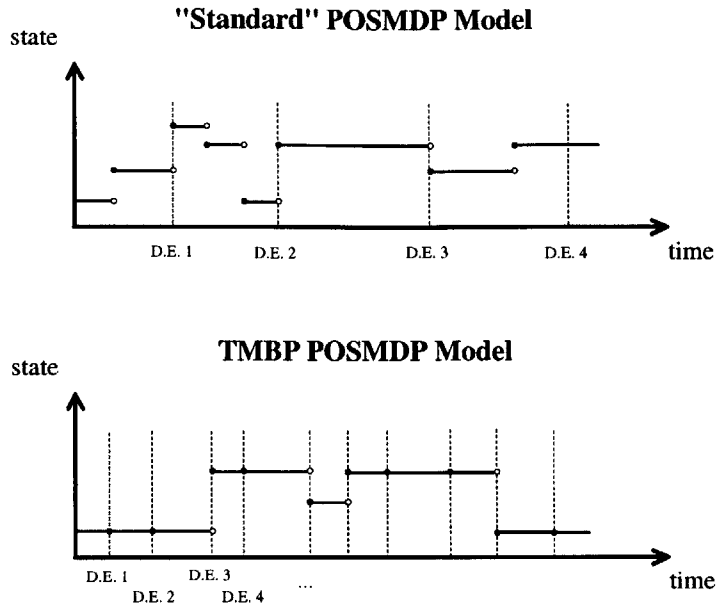

Figure 4-1: Illustration of the difference between the standard POSMDP model [58] and the variant defined in TMBP. "D.E." stands for "decision epoch", that is, a time at which a decision is made.

In both models, state transitions depend not only on the current state and action, but also on how long the system has been in the current state. The standard POSMDP model represents separately a transition probability function describing transitions at decision epochs only (independent of time) and a probability distribution of transition times for each state-action pair. The POSMDP plant model adopted for TMBP captures this time dependency explicitly in a single transition probability function, defined below, which specifies a probability associated with a particular transition between states and valuation of the clock variables. This definition of a single transition probability function is similar to the semi-Markov model considered by Lunze in [57], though his model does not consider partial observability in the system.

Another important distinction of the POSMDP model presented here is that it is factored. Other work has similarly leveraged factored representations of POMDPs, in which state is implicitly described by an assignment to some set of state variables [9, 32]. Key differences in this work's factored POSMDP model are the dependence of the transition function on time (in the form of clock values), and the com-

pact representation of the implemented model as a set of concurrent constraint-based automata (the factored approaches described in [9, 32] employ Dynamic Bayesian Networks as their compact representation).

4.1.1 Timed Plant Model as a Factored POSMDP

The semantic model of a plant defines a set of *variables* Π. Π is partitioned into *state variables* Π^s, *control variables* Π^c, *observable variables* Π^o, and *clock variables* Π^t. Variables in Π^s, Π^c, and Π^o each range over a finite domain. A *state* s is defined as an assignment to each variable in Π^s. An *observation* o assigns a value to each variable in Π^o. A *control action* μ assigns a value to each variable in Π^c. A *clock interpretation* ν assigns a value to each clock variable in Π^t. For $\delta \in \Re^+$, the clock interpretation that adds δ to the value of each clock variable in ν is denoted by $\nu + \delta$. A *full assignment* σ is defined as a set consisting of an assignment to each variable in Π, that is, $\sigma = \langle s, o, \mu, \nu \rangle$.

Formally, the plant is modeled as $\mathcal{P} = \langle \Sigma, \mathbb{T}, \mathbf{P}_\Theta, \mathbf{P}_\mathbb{T}, \mathbf{P}_\mathbb{O}, \mathbb{R} \rangle$. Σ is the set of all possible full assignments over Π. Σ_s, the projection of Σ onto variables in Π^s, is the set of all possible states, and Σ_t is the set of all possible clock interpretations over Π^t. $\mathbb{T}$ is a finite set of *transitions*, where each transition $\tau \in \mathbb{T}$ is a function $\tau : \Sigma \to \Sigma_s \times \Sigma_t$. Upon transitioning into a new state, some subset of the clock variables in Π_t are reset to zero (corresponding to the components which have changed modes as a result of this transition). Thus, for $\langle s', \nu' \rangle = \tau(\sigma)$, s' denotes the state obtained by applying transition τ to the full assignment σ, and ν' denotes the clock interpretation resulting from resetting the appropriate subset of clocks. The transition function $\tau^n \in \mathbb{T}$ models the system's nominal behavior, while all other transition functions model failure. $\mathbf{P}_\mathbb{T}$ associates with each transition τ and full assignment σ a probability $\mathbf{P}_\tau(\sigma)$. $\mathbf{P}_\tau(\sigma)$ is shorthand for $\mathbf{P}_\tau(s' \mid s, \mu, \nu)$, where s, μ, and ν are the state, control, and clock variable assignments in σ. $\mathbf{P}_\Theta(s_0)$ is the probability that the plant has initial state s_0. The reward for being in state s is $\mathbb{R}(s)$. The probability of observing o in state s is $\mathbf{P}_\mathbb{O}(o \mid s)$.

As defined above, the factored POSMDP plant model extends the factored POMDP

semantic model adopted by the untimed model-based programming paradigm [89, 90]. The main extensions are:

1. the definition of clock variables and clock interpretations,

2. the conditioning of transitions on clock interpretations (i.e., the conditioning of the transition probability on the current clock interpretation), and

3. the resetting of clock variables upon transitions.

It should be noted that the factored POSMDP plant model shares the key features of the factored POMDP plant model from untimed model-based programming. First, it captures nominal and various off-nominal system behaviors, by defining multiple possible transitions from each full assignment. Second, it is encoded compactly using concurrency and constraints (recall the factored plant model in Figure 3-3).

The semantic model of the plant as a factored POSMDP has a number of similarities with other real-time modeling formalisms, such as Timed Transitions Systems [39] and Timed Automata [2]. In particular, both of these models augment an untimed transition system with non-deterministic time constraints on the transitions, and define a complex system as a composition of concurrently-operating automata. Both models also adopt a dense model of time. The most important differences in the factored POSMDP model described above are: (1) the adoption of a probabilistic transition model (as opposed to the purely non-deterministic transitions in Timed Transition Systems and Timed Automata), (2) the distinction made between nominal and failure transitions, (3) the association of a single plant clock variable with each state variable (only one clock variable is necessary to capture the amount of time elapsed since the component transitioned into its current mode), and (4) the conditioning of transitions on system variable assignments rather than general event "labels."

4.1.2 Legal Trajectories of the Plant

In TMBP, an *interleaving* model of computation is adopted[1], where execution proceeds in *cycles*. Each cycle i consists of an instantaneous "discrete" event and a "continuous" phase in which time advances by some amount $\delta^{(i)}$. As far as the plant model is concerned, the discrete events correspond to transitions between plant states, which are assumed to occur instantaneously at absolute system times $t^{(0)}$, $t^{(1)}$, ... The plant maintains its state between these discrete event times; that is, state $s^{(i)}$ is assumed to hold in time interval $[t^{(i)}, t^{(i+1)})$. The time step $\delta^{(i)} = t^{(i+1)} - t^{(i)}$ is not necessarily constant from one cycle i to the next; it is determined by the amount of time required for the Timed Model-based Executive to complete one cycle of control sequencer and deductive controller operations.

Given a sequence of control actions $[\mu^{(0)}, \mu^{(1)}, \ldots]$, a *legal plant trajectory* is represented discretely by a sequence of states $[s^{(0)}, s^{(1)}, \ldots]$ and clock interpretations $[\nu^{(0)}, \nu^{(1)}, \ldots]$, such that:

1. $s^{(0)}$ is a valid initial plant state, that is, $\mathbf{P}_\Theta(s^{(0)}) > 0$;

2. $\nu^{(0)}$ is a valid initial clock interpretation, that is, the value of each clock in $\nu^{(0)}$ is zero;

3. each transition from $s^{(i)}$ to $s^{(i+1)}$ occurs at time $t^{(i+1)}$;

4. for each i, there is a full assignment $\sigma^{(\bullet i+1)} \in \Sigma$ which agrees with $s^{(i)}$ on assignments to variables in Π^s, with $\mu^{(i)}$ on assignments to variables in Π^c, and with $\nu^{(i)} + \delta^{(i)}$ on assignments to variables in Π^t. $\sigma^{(\bullet i+1)}$ captures the state and clock interpretation at time $t^{(i+1)} - \epsilon$ (i.e., just prior to the transition), where ϵ is infinitesimally small;

5. $\langle s^{(i+1)}, \nu^{(i+1)} \rangle = \tau(\sigma^{(\bullet i+1)})$, for some $\tau \in \mathbb{T}$ with $\mathbf{P}_\tau(\sigma^{(\bullet i+1)}) > 0$, where $\nu^{(i+1)}$ assigns zero to the newly reset clocks, and agrees with $\sigma^{(\bullet i+1)}$ on assignments

[1] Real-time modeling formalisms such as Timed Transition Systems [39] and Timed Automata [2] have previously adopted a similar interleaving model of concurrency; Henzinger, Manna and Pnueli [39] have shown that the interleaving model is an appropriate model for capturing most phenomena of interest occurring in the timed execution of real-time systems.

to all other clock variables; and

6. the trajectory is *non-zeno* [1], that is, time as captured in the clock interpretations never converges: infinitely many transitions do not occur in a finite interval of time.

A trajectory involving only the nominal transition τ^n is called a *nominal trajectory.* A *simple* trajectory does not repeat any state.

This definition of legal trajectories of a timed plant model is similar to the definitions of *computations* of a Timed Transition System [39], and *runs* of Timed Automata [2], in that it specifies initial conditions, satisfies all timing constraints on transitions, alternates state transition activities and time progress activities, and exhibits the non-zeno property.

4.2 Timed Control Program

In this section, the semantics of the second part of the timed model-based program, the *timed control program*, is discussed. A legal execution of a timed control program is defined in terms of a timed sequence of *control program locations*, which represent the "state" of the control program's execution at any given time, *configuration goals*, which provide the mechanism for goal-driven execution, and *control program clock interpretations*, which provide the mechanism for conditioning goals and activities on time constraints. The semantics of timed control programs builds on the semantics of untimed control programs [89, 90], by introducing the clock interpretations and conditioning the transitions between program locations on these clock interpretations.

4.2.1 Timed Control Program as a Deterministic Automaton

A set Π^t_{cp} of *clock variables* is defined for the control program (distinct from the plant clock variables defined in Section 4.1). Whereas plant clocks have the specific role of keeping track of how long each component has been in its current mode, control program clocks are more general-purpose: a control program clock variable is

initialized whenever a new time reference is needed to trigger time-delayed goals and activities in the control specification. Control program clocks measure the system time elapsed since their initialization.

Similar to the definition of plant clock interpretation, a *control program clock interpretation* ω assigns a value to each control program clock in Π^t_{cp}. Ω_{cp} is defined as the set of all possible clock interpretations over Π^t_{cp}. A clock x^t is defined to be *active* in execution cycle i if it was initialized in some earlier cycle. Conversely, if x^t has not been initialized prior to cycle i, it is defined to be *inactive*. To keep the semantics simple, the assumption is made that, once active, clocks cannot become inactive. Unlike the clocks associated with the plant model, control program clocks are never "reset" to zero: a new clock is initialized whenever a new time reference is needed. No loss of generality follows from this assumption – introducing new clocks as they are needed in the control program enables the specification of parallel activities triggered on independent time conditions.

Formally, a *timed control program* is a deterministic automaton $\mathcal{TCP} = \langle L_{cp}, \lambda_{cp}, \tau_{cp}, g_{cp}, \iota_{cp}, \Sigma_s, \Omega_{cp} \rangle$. L_{cp} is the set of program locations, where $\lambda_{cp} \in L_{cp}$ is the program's initial location. τ_{cp} is a transition function $\tau_{cp} : L_{cp} \times \Sigma_s \times \Omega_{cp} \rightarrow L_{cp}$. Transitions between program locations are conditioned on plant state estimates and control program clock interpretations. Each location has a corresponding set of *clock initializations* $\iota_{cp}(l) \subseteq \Pi^t_{cp}$, which is the set of clocks to be initialized upon transitioning to location l. Each location $l \in L_{cp}$ also has a corresponding *configuration goal* $g_{cp}(l) \subset \Sigma_s$, which is the set of plant goal states associated with location l.

4.2.2 Legal Executions of the Timed Control Program

Similar to the timed plant model, the timed control program changes locations instantaneously at absolute system times $t^{(0)}$, $t^{(1)}$, ..., and maintains its location between these discrete event times. Again, the time step $\delta^{(i)} = t^{(i+1)} - t^{(i)}$ is determined by the amount of time required for the Timed Model-based Executive to complete one cycle of control sequencer and deductive controller operations, and is not necessarily constant over i. Given a sequence of most-likely state estimates $[\hat{s}^{(0)}, \hat{s}^{(1)}, \ldots]$ of a plant

$\mathcal{P}$, a *legal execution* of a timed control program $\mathcal{TCP}$ is represented by a sequence of locations $[l^{(0)}, l^{(1)}, \ldots]$, configuration goals $[g^{(0)}, g^{(1)}, \ldots]$, and clock interpretations $[\omega^{(0)}, \omega^{(1)}, \ldots]$, such that:

1. $l^{(0)}$ is the initial program location λ_{cp};
2. $\omega^{(0)}$ is a valid initial clock interpretation, that is, $\omega^{(0)}(x^t) = 0$ for all clocks $x^t \in \Pi^t_{cp}$, and all clocks are initially inactive;
3. $\langle \omega^{(i)}, \omega^{(i+1)} \rangle$ represents a legal clock interpretation sequence, that is:
   - for each clock x^t that is inactive in cycle i, $\omega^{(i)}(x^t) = 0$,
   - for each x^t that is active in cycle i, $\omega^{(i)}(x^t) = \omega^{(i-1)}(x^t) + \delta^{(i-1)}$, where $\delta^{(i-1)} \in \Re^+$ is the same for all active clocks,
   - for each x^t that is initialized in cycle i (i.e., $x^t \in \iota_{cp}(l^{(i)})$), x^t is active for all cycles $j > i$;
4. $\langle l^{(i)}, l^{(i+1)} \rangle$ represents a legal control program transition, that is, $l^{(i+1)} = \tau_{cp}(l^{(i)}, \hat{s}^{(i)}, \omega^{(i)} + \delta^{(i)})$; and
5. $g^{(i)}$ represents a valid configuration goal, that is, $g^{(i)} = g_{cp}(l^{(i)})$.

The semantics of a timed control program can thus be considered a variant of Deterministic Timed Automata [2], with two key distinctions:

1. its execution is conditioned on the hidden state of a physical plant;
2. its locations assert configuration goals intended to operate on the hidden state of a physical plant.

4.3 Timed Model-Based Executive

A timed model-based program is executed by a *Timed Model-based Executive*, defined as a high-level *control sequencer*, coupled to a low-level *deductive controller* (see Figure 1-3). In this section, the semantics of each of these two executive modules is presented.

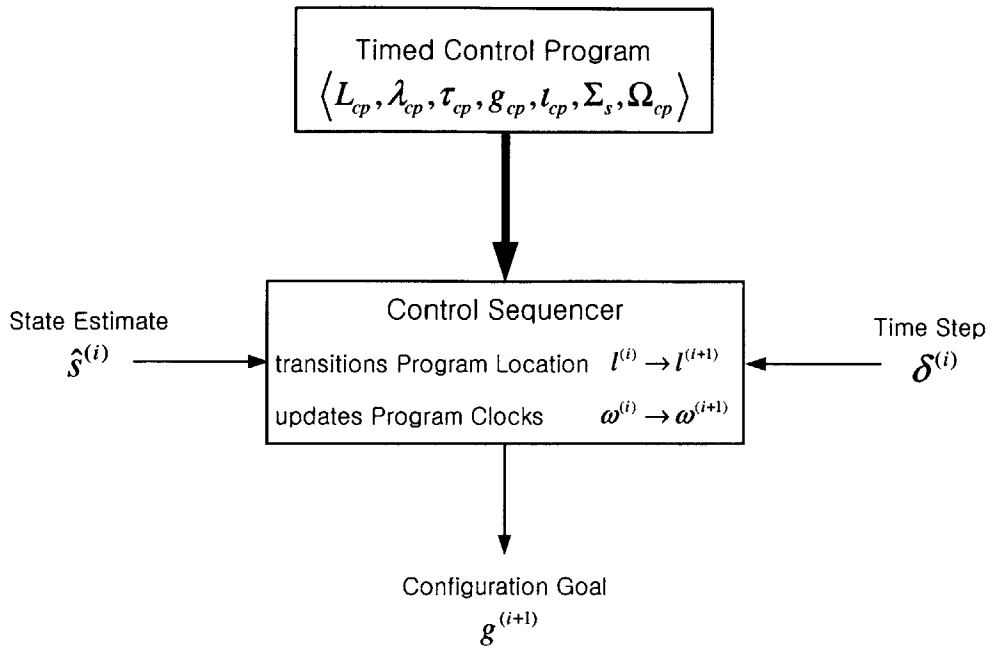

Figure 4-2: Block diagram showing inputs and outputs of the control sequencer.

4.3.1 Control Sequencer

The control sequencer's role is to direct the closed-loop goal-driven execution, by issuing the configuration goals specified in the timed control program. In each execution cycle, it takes as input a timed control program $\mathcal{TCP}$, the plant state estimate and the time step from the system clock, and it issues a configuration goal to the deductive controller (see Figure 4-2). More precisely, the control sequencer advances the control program from its current location $l^{(i)}$ to a new location $l^{(i+1)}$, by taking the transition enabled by the state estimate $\hat{s}^{(i)}$ and the clock interpretation $\omega^{(i)}$. It generates the configuration goal $g^{(i+1)}$ associated with the new program location $l^{(i+1)}$.

Formally, the semantics of the control sequencer can be described by the function CS (see Figure 4-3), which operates on the control program $\mathcal{TCP}$ defined in Section 4.2. Given an initial program location $l^{(0)}$, an initial clock interpretation $\omega^{(0)}$, and sequences of state estimates $[\hat{s}^{(0)}, \hat{s}^{(1)}, \ldots]$ and cycle time intervals $[\delta^{(0)}, \delta^{(1)}, \ldots]$, CS can be used to generate a legal execution of $\mathcal{TCP}$, i.e., it outputs legal sequences of program locations, clock interpretations and configuration goals (as defined in Section 4.2).

$$CS(\mathcal{TCP}, l^{(i)}, \hat{s}^{(i)}, \omega^{(i)}, \delta^{(i)}) \rightarrow \langle l^{(i+1)}, \omega^{(i+1)}, g^{(i+1)} \rangle ::$$

1. **Advance to new program location.**

$$l^{(i+1)} = \tau_{cp}(l^{(i)}, \hat{s}^{(i)}, \omega^{(i)} + \delta^{(i)})$$

2. **Update clock interpretations.**
For each clock $x^t \in \Pi^t_{cp}$:

$$\omega^{(i+1)} = \begin{cases} \omega^{(i+1)}(x^t) = 0 \text{ if } x^t \text{ is inactive;} \\ \omega^{(i+1)}(x^t) = \omega^{(i)}(x^t) + \delta^{(i)} \text{ if } x^t \text{ is active.} \end{cases}$$

x^t becomes active if $x^t \in \iota_{cp}(l^{(i+1)})$.

3. **Issue configuration goal.**

$$g^{(i+1)} = g_{cp}(l^{(i+1)}).$$

Figure 4-3: Control sequencer function *CS*.

4.3.2 Deductive Controller

The deductive controller's dual role is to (a) infer the system state based on observations from the sensors, and (b) issue control actions that achieve the configuration goals. In each execution cycle, it takes as input the plant model $\mathcal{P}$, the configuration goal from the control sequencer, the observation from the physical plant, and the time step from the system clock. It generates the most likely plant state estimate and an appropriate control action. This section presents the semantics of the deductive controller by defining semantics for its two distinct capabilities, *mode estimation* and *mode reconfiguration.*

Mode Estimation

The sequence of state estimates is generated by the deductive controller's mode estimation (ME) capability. ME is an online algorithm for tracking the likelihood of each possible plant state, given the plant model, the observations, the control actions, and the time elapsed since the last execution cycle. In each cycle, ME returns the most likely plant state as the current state estimate (see Figure 4-4).

Recall that previous work in model-based execution defined the semantic model of the plant as a factored POMDP [90]. In this case, ME is framed as an instance

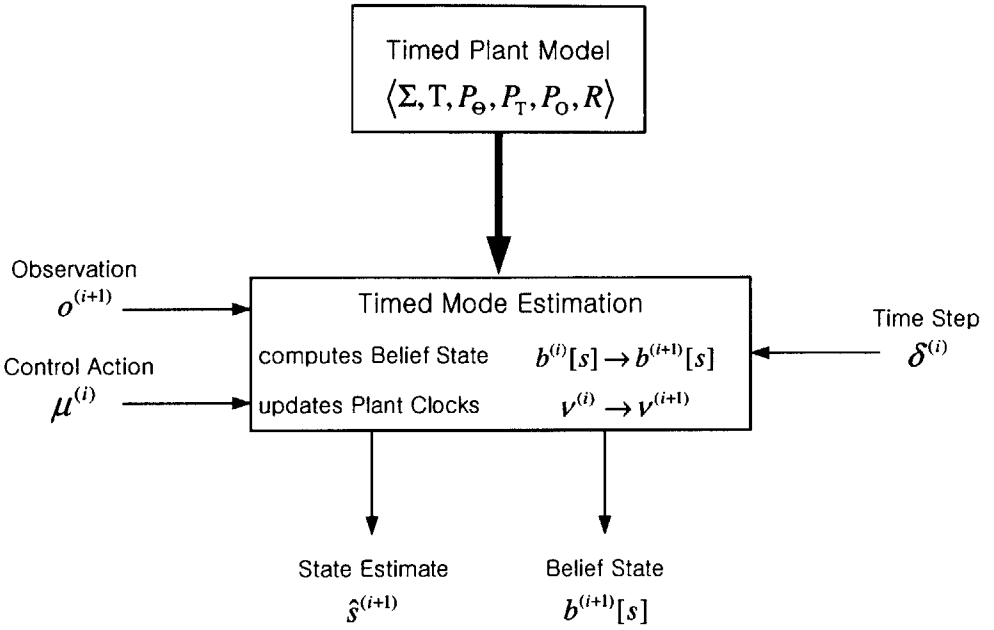

Figure 4-4: Block diagram showing inputs and outputs of mode estimation.

of POMDP *belief state update.* Belief state update computes the current *belief state*, that is, the probability associated with being in each state, conditioned on the control actions performed up to the *last* cycle, and the observations received up to the *current* cycle. Exploiting the Markov property, the belief state $b^{(i+1)}[s]$ at execution cycle $i+1$ is computed from the belief state and control actions at cycle i and observations at cycle $i+1$ using the following belief update equations:

$$p^{(\bullet i+1)}[s_k] = \sum_{j=1}^{n} p^{(i\bullet)}[s_j]\mathbf{P}_{\mathbb{T}}(s_k \mid s_j, \mu^{(i)})$$

$$p^{(i+1\bullet)}[s_k] = p^{(\bullet i+1)}[s_k]\frac{\mathbf{P}_{\mathbb{O}}(o^{(i+1)} \mid s_k)}{\sum_{j=1}^{n} p^{(\bullet i+1)}[s_j]\mathbf{P}_{\mathbb{O}}(o^{(i+1)} \mid s_j)}$$

where $p^{(\bullet i+1)}[s_k] \equiv p(s_k^{(i+1)} \mid o^{(0)}, \ldots, o^{(i)}, \mu^{(0)}, \ldots, \mu^{(i)})$, and $b^{(i+1)}[s_k] \equiv p^{(i+1\bullet)}[s_k] \equiv p(s_k^{(i+1)} \mid o^{(0)}, \ldots, o^{(i+1)}, \mu^{(0)}, \ldots, \mu^{(i)})$. $\mathbf{P}_{\mathbb{T}}(s_k \mid s_j, \mu^{(i)})$ is defined as the probability that $\mathcal{P}$ transitions from state s_j to state s_k, given control actions $\mu^{(i)}$. $\mathbf{P}_{\mathbb{O}}(o^{(i+1)} \mid s_k)$ is the probability that observation $o^{(i+1)}$ is received in state $s_k^{(i+1)}$. The initial belief state $b^{(0)}$ is computed based on $p^{(\bullet 0)}[s_k] = \mathbf{P}_\Theta(s_k)$.

Traditional belief state update associates a probability to each state in a *Trellis*

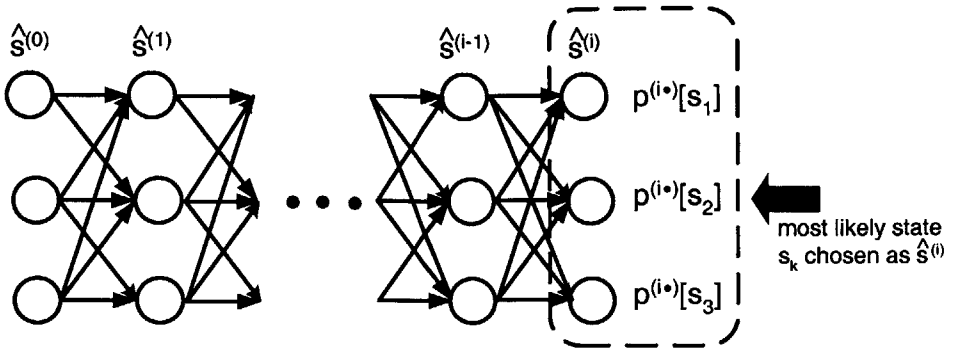

Figure 4-5: A Trellis diagram depicts the plant's possible state trajectories. Given the previous belief state, the latest control action and the latest observation, ME computes the current belief state, and selects the most likely state as its estimate for $\hat{s}^{(i)}$.

diagram, which enumerates all possible states at each time step and all transitions between states at adjacent times (Figure 4-5). For ME, the tracked state with the highest belief state probability is selected as the most likely state $\hat{s}^{(i)}$.

For the representation of the plant as a factored variant of POSMDP, traditional belief state update is not directly applicable: the semi-Markov nature of the component models implies that the evolution of a component's state depends not only on the current state and control actions taken, but also on the amount of time spent in the current state (as captured in the current clock interpretation). However, by considering the *system state* $\tilde{s}$ to be *the plant state augmented with the current clock interpretation*, i.e. $\tilde{s} = \langle s, \nu \rangle$, the factored POSMDP $\mathcal{P}$ can be converted to a factored POMDP $\widetilde{\mathcal{P}} = \langle \widetilde{\Sigma}, \widetilde{\mathbb{T}}, \widetilde{\mathbf{P}_\Theta}, \widetilde{\mathbf{P}_\mathbb{T}}, \widetilde{\mathbf{P}_\mathbb{O}}, \widetilde{\mathbb{R}} \rangle$, where:

- $\widetilde{\Sigma}$ is the set of full assignments, computed as the cross product $\widetilde{\Sigma_s} \times \Sigma_c \times \Sigma_o$, where $\widetilde{\Sigma_s}$ is the set of all possible system states $\tilde{s}$. Due to the inclusion of the real-valued clock interpretations into the state, $\widetilde{\Sigma_s}$ has an infinite domain, in contrast with the finite domain of the state space of $\mathcal{P}$. Note that, for each $\tilde{\sigma} = \langle \tilde{s}, o, \mu \rangle \in \widetilde{\Sigma}$, there is a one-to-one correspondence with $\sigma = \langle s, o, \mu, \nu \rangle \in \Sigma$ in the original factored POSMDP $\mathcal{P}$.

- $\widetilde{\mathbb{T}}$ is the set of transitions, where each transition $\tilde{\tau} \in \widetilde{\mathbb{T}}$ is a function $\tilde{\tau} : \widetilde{\Sigma} \rightarrow \widetilde{\Sigma_s}$. The transitions are derived from $\mathcal{P}$'s transition set $\mathbb{T}$, with the appropriate

clocks reset in the target system state. More precisely, for each $\tau \in \mathbb{T}$ in $\mathcal{P}$, a corresponding $\tilde{\tau} \in \widetilde{\mathbb{T}}$ is defined such that $\tilde{s}' = \tilde{\tau}(\tilde{\sigma})$, where $\tilde{\sigma}$ is the full assignment corresponding to σ in $\mathcal{P}$, and $\tilde{s}' = \langle s', \nu' \rangle = \tau(\sigma)$.

- $\widetilde{\mathbf{P}_\Theta}(\tilde{s})$ is the probability that system state $\tilde{s}$ is the initial system state. For each $\tilde{s} = \langle s, \nu \rangle$,

$$\widetilde{\mathbf{P}_\Theta}(\tilde{s}) = \begin{cases} \mathbf{P}_\Theta(s) \text{ if } \nu \text{ assigns zero to each plant clock;} \\ 0 \text{ otherwise.} \end{cases}$$

- $\widetilde{\mathbf{P}_\mathbb{T}}$ associates with each transition $\tilde{\tau}$ corresponding to τ in $\mathcal{P}$, and full assignment $\tilde{\sigma}$ corresponding to σ in $\mathcal{P}$, a probability $\widetilde{\mathbf{P}_\tau}(\tilde{\sigma}) = \mathbf{P}_\tau(\sigma)$.

- $\widetilde{\mathbf{P}_\mathbb{O}}(o \mid \tilde{s})$ is the probability of observing o in system state $\tilde{s}$. For each $\tilde{s} = \langle s, \nu \rangle$, $\widetilde{\mathbf{P}_\mathbb{O}}(o \mid \tilde{s}) = \mathbf{P}_\mathbb{O}(o \mid s)$.

- $\widetilde{\mathbb{R}}(\tilde{s})$ is the reward associated with system state $\tilde{s}$. For each $\tilde{s} = \langle s, \nu \rangle$, $\widetilde{\mathbb{R}}(\tilde{s}) = \mathbb{R}(s)$.

Now that the factored POSMDP $\mathcal{P}$ has been mapped to a factored POMDP $\widetilde{\mathcal{P}}$, the POMDP belief state update equations can be applied, to compute the likelihood of the system states in each execution cycle:

$$\begin{aligned} p^{(\bullet i+1)}[\tilde{s}_k] &= \sum_{j=1}^{n} p^{(i\bullet)}[\tilde{s}_j] \widetilde{\mathbf{P}_\mathbb{T}}(\tilde{s}_k \mid \tilde{s}_j, \mu^{(i)}) \\ p^{(i+1\bullet)}[\tilde{s}_k] &= p^{(\bullet i+1)}[\tilde{s}_k] \frac{\widetilde{\mathbf{P}_\mathbb{O}}(o^{(i+1)} \mid \tilde{s}_k)}{\sum_{j=1}^{n} p^{(\bullet i+1)}[\tilde{s}_j] \widetilde{\mathbf{P}_\mathbb{O}}(o^{(i+1)} \mid \tilde{s}_j)}, \end{aligned}$$

where the belief state $b^{(i+1)}[\tilde{s}_k] = p^{(i+1\bullet)}[\tilde{s}_k]$. The transition probability $\widetilde{\mathbf{P}_\mathbb{T}}(\tilde{s}_k \mid \tilde{s}_j, \mu^{(i)})$ is computed as the sum of $\widetilde{\mathbf{P}_\tau}(\tilde{\sigma})$ over transitions $\tilde{\tau}$, such that $\tilde{\sigma}$ includes the system state $\tilde{s}_j$ and the control action $\mu^{(i)}$, and $\tilde{\tau}(\tilde{\sigma}) = \tilde{s}_k$.

Once the likelihood of each possible system state has been computed, ME must extract from this belief state $b^{(i+1)}$ the probability of each possible plant state, to determine the most likely state estimate $\hat{s}^{(i+1)}$. The probability of each possible

plant state s is computed by summing the belief state probabilities associated with all system states $\tilde{s}^{(i+1)}$ that include plant state s. The plant state with the highest probability is returned by ME as $\hat{s}^{(i+1)}$. The belief state itself is needed for the deductive controller's mode reconfiguration capability, so it is also considered an output of ME, in Figure 4-4.

Even for an untimed plant model, the state space of the factored POMDP is very large, on the order of m^n, where n is the number of components in the system, and m is the average number of modes for each component. Various implementations of the untimed ME capability have been developed, which provide tractable approximations to this belief state update computation [90, 87, 83, 50]. By augmenting the state to include the current clock interpretations, the timed ME problem becomes even more computationally expensive due to the continuous nature of the system state space. Given the high level of reactivity required for the Timed Model-based Executive, it is necessary to approximate the belief state update process, such that only a limited number of the most likely state estimates are computed in each execution cycle. Section 6.2 provides more detail on belief state update for an implementation of the factored POSMDP semantic model, and on the approximations used to make the ME problem computationally tractable.

Mode Reconfiguration

The sequence of control actions is generated by the deductive controller's mode reconfiguration (MR) capability. MR provides an online algorithm for finding an optimal policy that achieves the configuration goal, given the plant model and the current belief state from ME (a policy π is a state-action mapping that specifies, for each state, an action to be taken). In each execution cycle, MR returns the first control action from the optimal policy (see Figure 4-6).

In the untimed model-based execution case, the semantics of MR maps to a *goal-directed decision theoretic planning* problem, based on the plant model expressed as a POMDP. In decision theoretic planning for POMDPs, the objective is to choose actions such that some measure of reward is maximized [46]. More precisely, decision

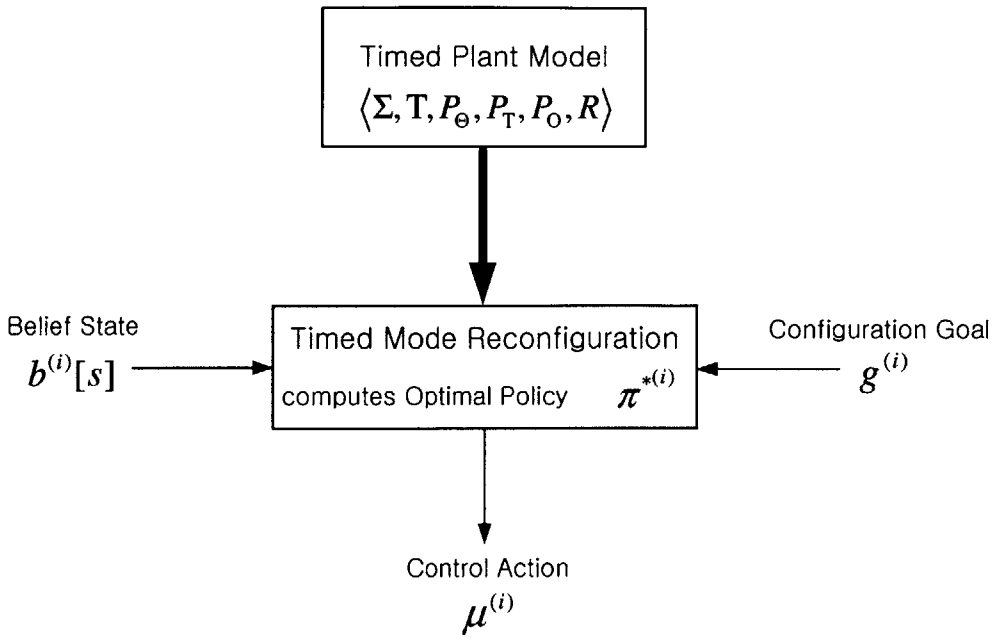

Figure 4-6: Block diagram showing inputs and outputs of mode reconfiguration.

theoretic planning computes an optimal policy π^* for the POMDP that maximizes the expected (possibly discounted) sum of reward over the finite discrete-time horizon of interest h:

$$\max_{\pi} E\left[\sum_{i=0}^{h} \gamma^i r_i\right].$$

The MR problem is described as goal-directed, because the specific objective of the planning problem is to find a policy that leads to a state that satisfies the given configuration goal. As such, the reward metric $R(s)$ for the MR planning problem is defined as a sum of a goal-specific reward function, which biases the solution toward states that achieve the specified configuration goal, and the state reward function $\mathbb{R}(s)$ built into the plant model POMDP, which biases the solution toward lower-cost policies among the set of policies that achieve the configuration goal.

The solution of the basic decision-theoretic planning problem (assuming the system is a Markov Decision Process (MDP) with fully-observable states) is obtained by

solving the Bellman optimality equations [66]:

$$V_i^*(s) = \max_{\mu} \left[R(s) + \gamma \sum_{s' \in \Sigma} \mathbf{P}_{\mathbb{T}}(s' \mid s, \mu) V_{i-1}^*(s') \right],$$

$$\pi_i^*(s) = \arg\max_{\mu} \left[R(s) + \gamma \sum_{s' \in \Sigma} \mathbf{P}_{\mathbb{T}}(s' \mid s, \mu) V_{i-1}^*(s') \right].$$

In these equations, the optimal value function in the i^{th} cycle, V_i^*, is defined inductively as the maximum of the sum of the immediate reward R and the discounted expected value of the remaining $(i-1)$ steps. The optimal policy for the i^{th} cycle, π_i^*, is defined in terms of the optimal value function V_{i-1}^* for the $(i-1)^{th}$ cycle. Common approaches for solving the Bellman equations for a MDP include *value iteration* [42] and *policy iteration* [36].

For a plant model modeled as a POMDP, the solution to the planning problem is obtained by solving the Bellman equations associated with the "belief MDP" $\langle \mathcal{B}, \mathcal{A}, \tau, \rho \rangle$ [46], where:

- $\mathcal{B}$, the set of belief states $b(s) = p^{(i\bullet)}[s]$ of the POMDP, defines the state space for the belief MDP;
- $\mathcal{A}$ is the set of actions for the belief MDP, corresponding to the set Σ_c of control actions for the POMDP;
- $\tau(b, \mu, b')$ is the probability that the belief state transitions from b in cycle i to b' in cycle $i+1$, given control action μ in cycle i. This probability is computed as follows:

$$\tau(b, \mu, b') = \sum_{o \in \Sigma_o} p(b' \mid b, \mu, o) p(o \mid b, \mu),$$

where

$$p(b' \mid b, \mu, o) = \begin{cases} 1 \text{ if } \mathit{BeliefUpdate}(\mathcal{P}, b, \mu, o) = b' \\ 0 \text{ otherwise;} \end{cases}$$

and

$$p(o \mid b, \mu) = \sum_{j=1}^{n} p^{(\bullet i+1)}[s_j] \mathbf{P}_{\mathbb{O}}(o \mid s_j).$$

- $\rho(b, \mu)$ is the reward function on belief states, computed from the original MR planning problem's reward metric:

$$\rho(b, \mu) = \sum_{s \in \Sigma_s} b(s) R(s)$$

Just as the belief update equations could not be applied directly to the states in the timed plant model, due to the model's semi-Markov nature, the Bellman optimality equations cannot be directly applied to the timed MR problem. However, using the same insight introduced for ME, augmenting the state with the clock interpretation transforms the factored POSMDP into a factored POMDP, which would enable the use of the POMDP decision theoretic planning approach described above. The resulting solution would be an optimal policy for the given timed plant model $\mathcal{P}$, the current configuration goal $g^{(i)}$, and the current belief state $b^{(i)}$.

However, it has been shown that solution of the decision theoretic planning problem associated with the "belief MDP" is generally intractable via exact dynamic programming algorithms, even for a standard POMDP model, due to the continuous nature of the belief state space [8, 46]. Indeed, it has been shown that finding the optimal policy for a finite-horizon POMDP is a PSPACE-complete problem [62]. Numerous approaches have been developed for finding approximate solutions, including for example Sondik's One-Pass algorithm [77] and the Witness algorithm [46]. These algorithms take the approach of computing approximations to the optimal value function, by exploiting properties of the belief state space. For the huge factored state spaces (exponential in the number of components) of interest in TMBP, this approach does not scale well [46].

Instead, this work follows the approach adopted in untimed model-based programming, by making key assumptions that allow MR to be performed reactively by dividing the planning problem into two steps: first, find a reachable goal state that satisfies the configuration goal and maximizes reward; second, find a (possibly suboptimal) sequence of control actions that lead from the current most-likely state to the maximum-reward goal state. These assumptions and the resulting implementation of

MR are described in Section 6.3.

4.4 Timed Model-based Program Execution

Now that the semantics of the timed model-based program and the various modules of the Timed Model-based Executive have been introduced, this section combines these semantic descriptions into an overall execution semantics for the timed model-based program.

Given a timed model-based program consisting of a timed plant model $\mathcal{P}$ and a timed control program $\mathcal{TCP}$, a sequence of cycle time intervals $[\delta^{(0)}, \delta^{(1)}, \ldots]$, and a sequence of observations $[o^{(0)}, o^{(1)}, \ldots]$, a *legal execution* of the timed model-based program is represented by sequences of state estimates $[\hat{s}^{(0)}, \hat{s}^{(1)}, \ldots]$ of $\mathcal{P}$, plant clock interpretations $[\nu^{(0)}, \nu^{(1)}, \ldots]$ of $\mathcal{P}$, program locations $[l^{(0)}, l^{(1)}, \ldots]$ of $\mathcal{TCP}$, program clock interpretations $[\omega^{(0)}, \omega^{(1)}, \ldots]$ of $\mathcal{TCP}$, configuration goals $[g^{(0)}, g^{(1)}, \ldots]$ of $\mathcal{TCP}$, and control actions $[\mu^{(0)}, \mu^{(1)}, \ldots]$, such that:

1. The initial conditions are valid, that is:

   - $\mathbf{P}_\Theta(\hat{s}^{(0)}) > 0$;
   - $l^{(0)}$ is the initial program location λ_{cp};
   - $\omega^{(0)}(x^t_{\mathcal{TCP}}) = 0$ for all program clocks $x^t_{\mathcal{TCP}}$; and
   - $\nu^{(0)}(x^t_{\mathcal{P}}) = 0$ for all plant clocks $x^t_{\mathcal{P}}$.

2. The sequences of program locations, program clock interpretations, and configuration goals correspond to a legal execution of $\mathcal{TCP}$, which is consistent with the semantics of the control sequencer (presented in Section 4.3.1).

3. If plant state $\hat{s}^{(i+1)}$ is the result of a nominal plant transition from $\hat{s}^{(i)}$, then $\hat{s}^{(i+1)}$ is the state resulting from taking $\mu^{(i)}$, the first action in an optimal policy that achieves configuration goal $g^{(i)}$, consistent with the semantics of MR (presented in Section 4.3.2).

4. Given the sequence of control actions, the sequences of state estimates and plant clock interpretations correspond to a legal trajectory of $\mathcal{P}$, which is consistent with the semantics of ME (presented in Section 4.3.2)

4.5 Summary

The semantics of both parts of the timed model-based program have been presented. The timed plant model is described as a variant of factored POSMDP; it extends untimed plant models with plant clocks, and specifies non-deterministic timed transitions between modes. It is distinguished from other real-time modeling formalisms, such as Timed Automata [2], in its constraint-based factored implementation, its adoption of a probabilistic transition model, its representation of semi-Markov behavior through a single plant clock variable for each state variable, and its conditioning of transitions on system variable constraints rather than general event "labels." The timed control program is specified as a deterministic automaton; it extends untimed control programs with program clocks, and enables transitions between locations to be conditioned on time. It differs from representations like Deterministic Timed Automata [2] in its mechanisms for conditioning on, and operating on, hidden states of the physical plant.

The semantics of the Timed Model-based Executive modules have also been specified. The control sequencer executes a timed control program and issues configuration goals for achievement by the deductive controller, which operates on the factored POSMDP model of the physical plant. The deductive controller provides a mode estimation capability, which performs a variant of belief state update, and a mode reconfiguration capability, which performs a variant of decision theoretic planning. The deductive controller semantics leverages the key insight that the plant state can be augmented with the plant clock interpretations, leading to a mapping from a semi-Markov to a Markov process. Finally, the semantic models for each of the executive modules have been composed into an overall execution semantics for timed model-based programs.

In the following two chapters of the thesis, one particular instance of a TMBP language and its corresponding executive are defined. In Chapter 5, an implementation of the control sequencer is presented, including the specification of a graphical model-based programming language, in the spirit of StateCharts [37], that allows engineers to specify desired state trajectories in the form of timed control programs. Chapter 6 presents an implementation of the deductive controller for the Timed Model-based Executive, including the specification of another graphical specification language, similar to Probabilistic Timed Automata [52], that provides the expressivity necessary to model physical plant behaviors. The practical importance of this instance of TMBP language is demonstrated in Chapter 7, on a representative spacecraft application.

Chapter 5

Control Sequencer

In this chapter, the control sequencer module of the implemented Timed Model-based Executive is described. The chapter begins with a description of the textual language RMPL, which is used to specify timed control programs for embedded systems. Executing a timed control program written in RMPL involves compiling it to a variant of hierarchical automata, called *Timed Hierarchical Constraint Automata (THCA)*, and then executing the automata in coordination with the deductive controller. In addition to providing a compact computational model for timed control programs, the THCA representation is a graphical specification language in its own right, similar to StateCharts [37]. THCA can be used directly by systems engineers in the design, analysis and verification of control specifications for embedded systems. After a formal definition of the THCA as a specific instance of the deterministic timed control program automaton presented in Section 4.2, the compilation of RMPL to THCA is discussed. Finally, the execution algorithm used by the control sequencer is presented, and its execution is illustrated on the timed control program for the Mars entry example introduced in Chapter 3.

5.1 The Reactive Model-based Programming Language

This section introduces RMPL, by first introducing its underlying constraint system, and then discussing the desired features of a TMBP language, in the context of a motivating example. Finally, the RMPL constructs used for writing timed control programs are introduced.

5.1.1 Constraint System: Propositional State Logic

A constraint system $\langle D, \models \rangle$ is a set of tokens D, closed under conjunction, together with an entailment relation $\models \subseteq D \times D$. The relation $\models$ satisfies the standard rules for conjunction (identity, $\wedge$ elimination, cut and $\wedge$ introduction), as defined in [73]. RMPL currently supports *propositional state logic* as its constraint system. Constraints can be of two types: *state constraints* and *clock constraints.* In the case of state constraints, propositions take the form $(x^s = v)$, where variable x^s ranges over a finite domain $\mathbb{D}[x^s]$; in the case of clock constraints, propositions take the form $(x^t \ ineq \ t)$, where clock variable x^t ranges over $\Re^+$, and $ineq \in \{<, >, \leq, \geq\}$. A proposition can have a truth assignment of true or false. Propositions are composed into formulae using the standard logical connectives: and $(\wedge)$, or $(\vee)$, and not $(\neg)$. The constants **True** and **False** are also valid constraints. A constraint is *entailed* if it is implied by the conjunction of the plant model and the most likely current state of the physical plant; otherwise, it is *not entailed.* Entailment is denoted by simply stating the constraint. Non-entailment is denoted by using an overbar. Note that non-entailment of constraint c (denoted $\bar{c}$) is *not* equivalent to entailment of the negation of c $(\neg c)$; the current knowledge of plant state may not imply c to be true or false.

In specifying an RMPL control program, the objective is to specify the desired behavior of the plant by stating constraints that, when made true, will cause the plant to follow a desired state trajectory. State assertions are specified as constraints

on plant state variables that should be made true. RMPL's model of interaction is in contrast to that of embedded programming languages, like Esterel [6], and concurrent constraint programming languages, like TCC [74]. Embedded programming languages interact with the program memory, sensors and control variables, but not directly with the plant state. For example, Esterel interacts by emitting and detecting signals. In contrast, RMPL control programs *ask* constraints on plant state variables, and request that specified constraints on state variables be *achieved* (as opposed to *tell*, which asserts that a constraint **is** true). State assertions in RMPL control programs are treated as *achieve* operations, while state condition tests are *ask* operations. The definition of *achieve* constraints distinguishes TMBP from concurrent constraint programming, where the language interacts with a constraint store strictly by *telling* and *asking* constraints. In TMBP, the constraint store is replaced by a deductive controller; the language interacts with the deductive controller by *asking* constraints on the system state, and issuing *achieve* constraints as goals on the system state (recall Figure 2-1). It should be noted that, although RMPL provides the flexibility of asserting any form of propositional logic constraint on plant variables, the current implementation of the Timed Model-based Executive handles only assertions of constraints that are conjunctions of state variable assignments.

5.1.2 RMPL Control Programs

To motivate RMPL's constructs, another example control program is considered, corresponding to an important type of critical mission sequence: the post-launch deployment sequence of a spacecraft. This sequence is roughly based on the Chandra X-Ray Telescope Activation Sequence Flowchart, included in Appendix A. This example assumes the spacecraft is launched into low Earth orbit by the Space Shuttle (as was the case for the Chandra observatory). It also assumes the spacecraft is attached to an Inertial Upper Stage (IUS), which serves to boost it into a higher Earth orbit, after it has been ejected from the Shuttle's payload bay. The IUS communicates with the spacecraft across a data interface, and is assumed to have its own processor, attitude control system (ACS), propulsion system, power system, etc...

The control program for spacecraft deployment is written from the point of view of the spacecraft processor, such that information from the IUS, which takes the form of simple binary signals sent across the interface, are considered external events. Once received, these signals trigger a flag to be set in the spacecraft's onboard memory. Because the spacecraft has no observability into the internal state of the IUS, it does not have access to the actual physical states that cause the signals to be issued. Thus, these flag settings represent important "indirect observation" states that drive the execution of the SpacecraftDeploy control program, shown in Figure 5-1.

SpacecraftDeploy proceeds as follows. Once the spacecraft (attached to the IUS) is notified that it has separated from the shuttle (line 3), it concurrently warms up its two IMUs and its Reaction Control System (RCS) thrusters (lines 4-8). It then waits to receive the next signal from the IUS (lines 10-14). In the nominal execution situation, the next signal triggers the flag indicating that the spacecraft can deploy its Low Gain Antenna (LGA) (line 10); in this case, the spacecraft executes the DeployLGA subprogram (line 11). On the other hand, if the next IUS signal triggers the Solar Array (SA) deployment flag first, the non-critical LGA deployment process is bypassed in favor of the DeploySolarArrays process (line 14).

The DeployLGA process involves several sequential steps:

1. it turns on at least one of the redundant Solid State Recorder (SSR) devices, to begin storing onboard telemetry (lines 21-24);

2. its communication subsystem begins downlinking a beacon signal (line 25);

3. it turns on the primary IMU, using the second IMU as a backup in the case of failure (lines 26-27); and

4. finally, it deploys its LGA (line 28).

Upon completion of this subprogram, the spacecraft waits for receipt of the Solar Array (SA) deployment signal from the IUS, if it hasn't already been received (line 12).

The DeploySolarArrays subprogram performs the following steps:

```
1   SpacecraftDeployment():: {
2     do {
3       when (shuttle = separated)  donext {
4         {
5           IMU_1 = warmed,
6           IMU_2 = warmed,
7           RCS = warmed
8         };
9         {
10          when ((LGA_deploy_flag = set) ∧ (¬(SA_deploy_flag = set)))  donext {
11            DeployLGA();
12            when (SA_deploy_flag = set)  donext DeploySolarArrays()
13          },
14          when ((SA_deploy_flag = set) ∧ (¬(LGA_deploy_flag = set)))  donext DeploySolarArrays()
15        }
16      }
17    }  watching ((spacecraft_deployment = success) ∨ (spacecraft_deployment = failure))
18  }
19
20  DeployLGA():: {
21    do {
22      SSR_A = on,
23      SSR_B = on
24    }  watching ((SSR_A = on) ∨ (SSR_B = on));
25    downlink = beacon;
26    do (IMU_1 = on)  watching (IMU_1 = failed);
27    if (IMU_1 = failed)  thennext (IMU_2 = on);
28    LGA = deployed
29  }
30
31  DeploySolarArrays():: {
32    t1 = 0;
33    unless ((SSR_A = on) ∨ (SSR_B = on))  thennext {
34      do {
35        SSR_A = on,
36        SSR_B = on
37      }  watching ((SSR_A = on) ∨ (SSR_B = on));
38    };
39    downlink = nominal;
40    unless ((IMU_1 = on) ∨ (IMU_2 = on))  thennext {
41      do (IMU_1 = on)  watching (IMU_1 = failed);
42      if (IMU_1 = failed)  thennext (IMU_2 = on);
43    };
44    do {
45      SA_1 = deployed,
46      SA_2 = deployed
47    }  watching ((SA_1 = failed) ∨ (SA_2 = failed));
48    {
49      if ((SA_1 = failed) ∨ (SA_2 = failed))  thennext (spacecraft_deployment = failure),
50      if ((SA_1 = deployed) ∧ (SA_2 = deployed))  thennext {
51        when (t1 ≥ 19 mins)  donext {
52          ACS = active;
53          when (t1 ≥ 22 mins)  donext {
54            power_switch = engage-onboard-batts;
55            when (IUS = separated)  donext (spacecraft_deployment = success)
56          }
57        }
58      }
59    }
60  }
```

Figure 5-1: RMPL control program for the Spacecraft Deployment procedure. In RMPL programs, comma delimits parallel processes and semicolon delimits sequential processes.

1. first, it initializes a clock variable, to keep track of the amount of time elapsed since the setting of the SA_deploy_flag (line 32);

2. in case the DeployLGA process was bypassed, it turns on at least one of the SSRs (lines 33-38);

3. it puts its communication subsystem into its nominal telemetry downlinking mode (line 39);

4. again, if the DeployLGA process was bypassed, it activates the primary (or backup) IMU (lines 40-43);

5. it then initiates the SA deployment process (lines 44-47), to be preempted if the deployment of either SA is determined to have failed (deployment of a SA will only fail after attempting recovery actions built into the SA subsystem model);

6. in the case of a SA deployment failure, failure of the entire SpacecraftDeploy sequence is flagged (line 49), resulting in preemption of the sequence (line 17);

7. in the case of nominal deployment of both SAs (line 50), the sequence waits for the active clock to indicate that 19 minutes have passed since the setting of the SA_deploy_flag (line 51) - this delay captures the time required for the IUS to null out any residual spacecraft rotations;

8. after 19 minutes, the spacecraft's own ACS is activated, in anticipation of the separation from the IUS, which has been providing attitude control for the spacecraft/IUS system (line 52);

9. after 22 minutes have elapsed since the setting of the SA_deploy_flag (line 53), capturing the time needed for the IUS to complete its pre-separation maneuvers, the power system switches from IUS-supplied power to power supplied by the spacecraft's onboard batteries (line 54); and

10. the spacecraft waits until the separation of the IUS via the successful firing of the pyro latches responsible for breaking the connection with the IUS, at which

point it confirms success of the full SpacecraftDeploy sequence (line 55) and terminates execution (line 17).

Desiderata

SpacecraftDeploy highlights six design features for RMPL. First, the program exploits full concurrency, by intermingling sequential and parallel threads of execution. For example, the IMUs and RCS are issued warming goals in parallel (lines 5-7), while deployment of the LGA and waiting for the SA deployment flag to be set are performed in sequence (lines 11-12). Second, it involves conditional execution, such as turning on the redundant backup IMU if the primary IMU fails to turn on (line 27). Third, it involves iteration; for example, "**when** (shuttle = separated) **donext** ..." (line 3) says to iteratively test until the shuttle state is determined to be separated, and then to proceed. Fourth, the program involves preemption; for example, "**do** (IMU_1 = on) **watching** (IMU_1 = failed)" (line 26) tries to turn on the primary IMU, but interrupts this effort as soon as the watched condition (IMU_1 = failed) is entailed. These first four features are common to most synchronous reactive programming languages.

The fifth feature of RMPL, which distinguishes the model-based programming paradigm, is the ability to reference hidden states of the physical plant within assertions and guards. For instance, "**if** (IMU_1 = failed) **thennext** (IMU_2 = on)" (line 27) uses constraints on the hidden state of an IMU in both a condition check (IMU_1 = failed) and an assertion (IMU_2 = on). The state of an IMU is considered hidden because it cannot be directly observed and commanded: the state must be inferred based on observations of electric currents through the IMU electronics, and can only be controlled via flight computer commands that pass through a communication bus and IMU driver electronics, for example.

The sixth and defining feature of RMPL, as far as the TMBP paradigm is concerned, is its use of time constraints to implicitly capture knowledge about the state of the spacecraft or its environment. For example, the spacecraft waits for 19 minutes to have passed since receipt of the SA deployment signal from the IUS before activating its own ACS; this allows the IUS enough time to control the residual rotations

in the unmodeled attitude state of the IUS/spacecraft system.

5.1.3 RMPL Language Specification

RMPL is a modular, constraint-based language that is closely related to other synchronous programming languages, such as Esterel [6], and constraint programming languages, such as TCC [74]. The RMPL language is specified by first introducing a small set of primitives for constructing timed control programs. These primitives are fully orthogonal, that is, they may be nested and combined arbitrarily. To make the language usable, a variety of derived combinators are defined on top of these primitives, such as those used in the EntrySequence and SpacecraftDeployment timed control programs (see Figures 3-2 and 5-1).

In the following discussion, lowercase letters, like c, are used to denote constraints, and uppercase letters, like A and B, are used to denote well-formed RMPL expressions. An RMPL expression is specified by the following grammar in Backus-Naur Form:

$$
\begin{aligned}
\textit{expression} &\rightarrow \textit{assertion} \mid \textit{combinator} \mid \textit{prgm\_invocation} \\
\textit{combinator} &\rightarrow A \text{ maintaining } c \mid \text{do } A \text{ watching } c \mid \\
&\quad\ \text{if } c \text{ thennext } A \mid \text{unless } c \text{ thennext } A \mid \\
&\quad\ A,B \mid A;B \mid \text{always } A \\
\textit{prgm\_invocation} &\rightarrow \textit{program\_name(arglist)}
\end{aligned}
$$

where an *assertion* is either a clock initialization of the form $(clock = 0)$ or a state constraint to be achieved. Note that procedure calls are specified as RMPL program invocations, in which *program_name* corresponds to another specified timed control program. The *arglist* used in a program invocation corresponds to a (possibly empty) list of parameters defined for the program. Within the invoked procedure, each parameter is replaced by the appropriate argument in *arglist*.

Primitive Combinators

RMPL provides standard primitive constructs for conditional branching, preemption, iteration, and concurrent and sequential composition. In general, RMPL constructs are conditioned on the current state of the physical plant and the current values of active clocks (that is, clocks that have been initialized); they can act on the plant state and/or initialize new clocks in the next time instant.

g: If g is an *achieve* constraint on state variables of the physical plant, this expression asserts that the plant should progress toward a state that entails constraint g. If g is an initialization of a clock variable, this expression activates the corresponding clock. This is the basic construct for affecting the plant's hidden state and starting new clocks.

A **maintaining** c: This expression executes expression A, while ensuring that constraint c is maintained true throughout. c is an *ask* constraint on active clock variables or state variables of the physical plant. If c is not entailed at any instant, then the execution thread terminates immediately. This is the basic construct for preemption by non-entailment.

do A **watching** c: This expression executes expression A, but if constraint c becomes entailed by the most likely plant state at any instant, it terminates execution of A in that instant. c is an *ask* constraint on active clock variables or state variables of the physical plant. This is the basic construct for preemption by entailment.

if c **thennext** A: This expression starts executing RMPL expression A in the next instant, if the most likely current plant state entails c. c is an *ask* constraint on active clock variables or state variables of the physical plant. This is the basic construct for conditionally branching upon entailment of the plant's hidden state and clock constraints.

unless c **thennext** A: This expression starts executing RMPL expression A in the next instant if the current theory does *not* entail c. c is an *ask* constraint on

active clock variables or state variables of the physical plant. This is the basic construct for conditionally branching upon non-entailment of the plant's hidden state and clock constraints.

A , B: This expression concurrently executes RMPL expressions A and B, starting in the current instant. It is the basic construct for forking processes.

A ; B: This is the sequential composition of RMPL expressions A and B. It performs A until A is finished, then it starts B.

always A: This expression starts expression A at each instant of time, for all time. This is the only iteration primitive needed, since finite iteration can be achieved by using a preemption construct to terminate an **always**.

Derived Combinators

The previously described primitive combinators cover the six desired design features. They can be used to implement a rich set of derived combinators, some of which are used in the EntrySequence and SpacecraftDeployment examples.

Some useful derived combinators are listed as follows:

next A: This expression starts executing expression A in the next instant. It is equivalent to:
{**if** *true* **thennext** A}.

if c **thennext** A_{then} **elsenext** A_{else}: This extends **if** c **thennext** A. Expression A_{else} is executed starting in the next instant if c is *not* entailed by the most likely current state. c is an *ask* constraint on active clock variables or state variables of the physical plant. This expression is equivalent to:
{**if** c **thennext** A_{then}, **unless** c **thennext** A_{else}}.

when c **donext** A: This is a temporally extended version of **if** c **thennext** A. It waits until constraint c is entailed by the most likely plant state, then starts executing A in the next instant. c is an *ask* constraint on active clock variables

or state variables of the physical plant. This expression is equivalent to: {**do always** *true* **watching** $c; A$}.

whenever c **donext** A: This is an iterated version of **when** c **donext** A. For every instant in which constraint c holds for the most likely state, it starts program A in the next instant. c is an *ask* constraint on active clock variables or state variables of the physical plant. This expression is equivalent to: {**always if** c **thennext** A}.

The primitive and derived RMPL constructs are used to encode model-based control programs. This subset is sufficient to implement most of the control constructs of the Esterel language [6]. A mapping between key Esterel constructs and analogous expressions in RMPL was presented in [44]. Note that RMPL can also be used to encode the probabilistic transition models capturing the behavior of the plant components. The additional constructs required to encode such models are defined in [87]. The plant model is further defined in Chapter 6.

5.2 Control Programs as Timed Hierarchical Constraint Automata

Engineers generally prefer to use visual representations of system specifications over textual encodings. For this reason, StateCharts and similar formalisms have become fairly standard tools in the design and analysis of embedded systems. The TMBP paradigm follows this trend, by adopting the THCA graphical representation for its timed control programs (e.g., Figure 3-4). Execution of these automata is performed by the control sequencer in coordination with the deductive controller. This section defines THCA as a specific instance of the deterministic timed control program automaton presented in Section 4.2. In addition to providing a visual programming paradigm, THCA are the computational model for timed control programs written in RMPL. The compilation from RMPL to THCA is also discussed in this section.

5.2.1 Timed Hierarchical Constraint Automata

This section provides a formal definition of the THCA language. In the following discussion, the "states" of a THCA are called *locations*, to avoid confusion with the physical plant state. The overall state of the program at any instant of time corresponds to a set of "marked" THCA locations. A THCA has seven main attributes:

1. it composes sets of concurrently operating automata;

2. each location is labeled with a state constraint, called a *goal constraint*, which the physical plant must immediately begin moving towards, whenever the automaton marks that location;

3. each location is labeled with a time constraint, called a *clock initialization*, which initializes a set of clock variables upon transitioning into the location;

4. each location is labeled with a constraint on clock and state variables, called a *maintenance constraint*, which must hold for that location to remain marked;

5. transitions between locations are conditioned on time and hidden state;

6. automata are arranged in a hierarchy – a location of an automaton may itself be an automaton, which is invoked when marked by its parent. This enables the initiation and termination of complex concurrent and sequential behaviors; and

7. each transition may have multiple target locations, allowing an automaton to have several locations marked simultaneously. This enables a compact representation for iterative behaviors, as illustrated in the Mars entry example (Figure 3-4).

THCA extend the Hierarchical Constraint Automata (HCA) defined in the context of model-based programming [86, 87, 90], to allow for the initialization of clock variables (feature #3), the possibility of preemption based on time constraints (feature #4), and the conditioning of transitions on time constraints (feature #5).

The THCA graphical language adopts various notions from real-time modeling formalisms like Timed Automata [2], including clock variables, clock interpretations, clock initializations, and timing constraints. However, whereas THCA are intended to provide a framework for executable specification of embedded control programs that run "in the loop" as part of a system's real-time control system, formal system specification languages are intended to provide a framework for off-line formal verification and model checking. This fundamental difference in intent leads to some key differences in the language, such as THCA's adoption of a hierarchical computational model. It should be noted that hierarchical encodings form the basis for embedded reactive languages like Esterel [6], StateCharts [37], and Timed StateCharts [47]. Distinctive features of a THCA are its compact encoding, and its use of constraints on plant state, in the form of goal constraints and maintenance constraints.

Formally, a THCA $\mathcal{A}$ is a tuple $\langle \Sigma, \Theta, \Pi, \mathbb{G}, \mathbb{I}, \mathbb{M}, \mathbb{T} \rangle$, where:

- Σ is a set of *locations*, partitioned into *primitive locations* Σ_p and *composite locations* Σ_c. Each composite location corresponds to another hierarchical constraint automaton. The set of *subautomata* of $\mathcal{A}$ is defined as the set of locations of $\mathcal{A}$, and locations that are descendants of the composite locations of $\mathcal{A}$:

$$subaut(\mathcal{A}) = \Sigma(\mathcal{A}) \cup \bigcup \{ subaut(\sigma^c) \mid \sigma^c \in \Sigma_c(\mathcal{A}) \}.$$

- $\Theta \subseteq \Sigma$ is the set of *start locations* of $\mathcal{A}$.

- Π is a set of variables, partitioned into Π^d and Π^t:

  - Π^d is the set of discrete plant state variables. Each $x^d \in \Pi^d$ ranges over a finite domain $\mathbb{D}[x^d]$. $\mathbb{C}[\Pi^d]$ denotes the set of all finite-domain constraints over variables in Π^d of the form $\lambda \triangleq TRUE \mid FALSE \mid (x^d = v) \mid \neg\lambda_1 \mid \lambda_1 \wedge \lambda_2 \mid \lambda_1 \vee \lambda_2$, where $v \in \mathbb{D}[x^d]$.

  - Π^t is the set of clock variables. Each $x^t \in \Pi^t$ ranges over $\Re^+$. $\mathbb{C}_{time}[\Pi^t]$ is the set of all possible clock variable constraints over Π^t of the form $\lambda \triangleq TRUE \mid FALSE \mid (x^t \leq t_1) \mid (x^t \geq t_1) \mid (x^t < t_1) \mid (x^t > t_1) \mid$

$\lambda_1 \wedge \lambda_2 \mid \lambda_1 \vee \lambda_2$, where $t_1 \in \Re^+$. $\mathbb{C}_{init}[\Pi^t]$ is the set of all possible clock initializations over Π^t of the form $\lambda \triangleq (x^t = 0) \mid \lambda_1 \wedge \lambda_2$.

- $\mathbb{G} : \Sigma_p \rightarrow \mathbb{C}[\Pi^d]$, associates with each primitive location $\sigma^p \in \Sigma_p$ a finite-domain constraint $\mathbb{G}(\sigma^p)$ that the plant progresses towards whenever σ^p is marked. $\mathbb{G}(\sigma^p)$ is called the *goal constraint* of σ^p.

- $\mathbb{I} : \Sigma_p \rightarrow \mathbb{C}_{init}[\Pi^t]$, associates with each primitive location $\sigma^p \in \Sigma_p$ a set of initializations of clock state variables $\mathbb{I}(\sigma^p)$ that are performed when σ^p is initially marked. $\mathbb{I}(\sigma^p)$ is called the *clock initialization* of σ^p.

- $\mathbb{M} : \Sigma \rightarrow \mathbb{C}[\Pi^d] \times \mathbb{C}_{time}[\Pi^t]$, associates with each location $\sigma \in \Sigma$ a pair $\mathbb{M}(\sigma) = (\mathbb{M}^d(\sigma), \mathbb{M}^t(\sigma))$ consisting of a discrete state constraint and a clock constraint that must both hold at the current instant for σ to be marked. $\mathbb{M}(\sigma)$ is called the *maintenance constraint* of σ.

- $\mathbb{T} : \Sigma \times \mathbb{C}[\Pi^d] \times \mathbb{C}_{time}[\Pi^t] \rightarrow 2^\Sigma$ associates with each location $\sigma \in \Sigma$ a transition function $\mathbb{T}(\sigma)$. Each $\mathbb{T}(\sigma) : \mathbb{C}[\Pi^d] \times \mathbb{C}_{time}[\Pi^t] \rightarrow 2^\Sigma$, specifies a *set* of locations to be marked in execution cycle $i + 1$, given appropriate assignments to Π in cycle i.

At execution cycle i, the state of a THCA is the set of marked locations $m^{(i)} \subseteq \Sigma$, called a *marking.* A marking represents the current state of multiple concurrent threads of execution. $\mathfrak{M}$ denotes the set of possible markings, where $\mathfrak{M} \subseteq 2^\Sigma$. Goal constraints $\mathbb{G}(\sigma^p)$ may be thought of as abstract set points, representing a set of states that the plant must evolve towards when σ^p is marked. Maintenance constraints $\mathbb{M}(\sigma)$ may be viewed as representing monitored constraints that must be maintained in order for execution to progress towards achieving any goal constraints specified within σ.

In the graphical representation of THCA, primitive locations are represented as circles (locations 2-7 and 9-12, in Figure 3-4), while composite locations are represented as rectangles (locations 1 and 8 in Figure 3-4). Clock initializations, goal constraints and maintenance constraints are written within the corresponding locations, with maintenance constraints preceded by the keyword "maintain". Looking at

Figure 3-4, the assignment *"t1=0"* in location 3 is an example of a clock initialization, and the assignment *"nav=inertial"* in location 5 is an example of a goal constraint. Maintenance constraints can be of the form $\models c$ or $\not\models c$, for some $c \in \mathbb{C}[\Pi^d] \times \mathbb{C}_{time}[\Pi^t]$. For convenience, c is used in the THCA diagrams to denote the constraint $\models c$, and $\bar{c}$ is used to denote the constraint $\not\models c$ (e.g., *"MAINTAIN* $\overline{\textit{entry=initiated}}$*"* for location 8, in Figure 3-4). Maintenance constraints associated with composite locations are assumed to apply to all subautomata within the composite location. When either a goal or a maintenance constraint is not specified, it is taken to be implicitly **True**.

Transitions are conditioned on constraints that must be satisfied by the conjunction of the plant model, the most likely estimated state of the plant, and the current clock interpretation. For each location σ, the transition function $\mathbb{T}(\sigma)$ is represented as a set of transition pairs (l, σ'), where $\sigma' \in \Sigma$, and l is a label (also known as a *guard condition*) of the form $\models c$ (denoted c) or $\not\models c$ (denoted $\bar{c}$), for some $c \in \mathbb{C}[\Pi^d] \times \mathbb{C}_{time}[\Pi^t]$. This corresponds to the traditional representation of transitions as labeled arcs in a graph, where σ and σ' are the source and target of an arc with label l. In Figure 3-4, the guard on the transition from location 7 to itself is the time constraint *"t1<4 min"*, and the guard on the transition between locations 11 and 12 is the state constraint *"att=entry-orient"*. Again, if no label is indicated, it is implicitly **True**.

The THCA encoding has four properties that distinguish it from the hierarchical automata employed by other reactive embedded languages [6, 37, 47]. First, multiple transitions may be simultaneously traversed. This permits a compact encoding of the state of the automaton as a set of markings. Second, transitions are conditioned on what can be deduced about the plant state, not just what is explicitly observed or assigned. This provides a simple, but general, mechanism for reasoning about the plant's hidden state. Third, transitions can be enabled based on lack of information, that is, non-entailment of a constraint. This allows default executions to be pursued in the absence of better information, enabling advanced preemption constructs. Finally, locations assert goal constraints on the plant state. This allows the hidden state of the plant to be controlled directly.

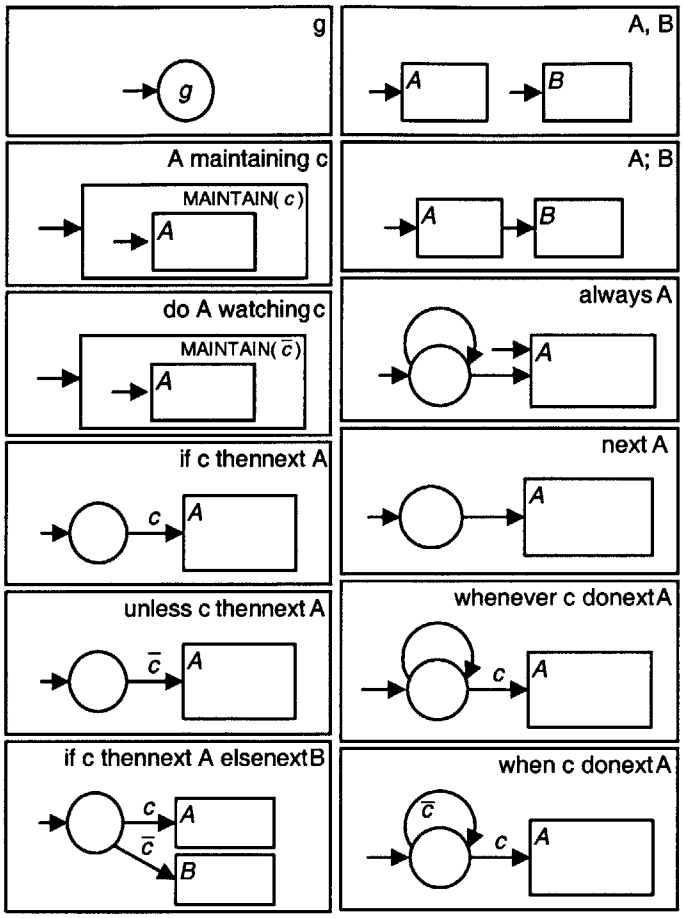

Figure 5-2: Corresponding THCA for various RMPL constructs.

5.2.2 Compiling RMPL to THCA

Each RMPL construct maps to a THCA as shown in Figure 5-2. The derived combinators are definable in terms of the primitives, but for efficiency they are mapped directly to THCA as well. As before, lowercase letters denote constraints expressed in propositional state logic. Uppercase letters denote well-formed RMPL expressions, each of which maps to a THCA.

To illustrate compilation, consider the RMPL code for Mars entry, shown earlier in Figure 3-2. The RMPL compiler converts this to the corresponding THCA, shown in Figure 3-4.[1] The next section describes the algorithm used by the Timed Model-

[1]Note that the compilation process takes place offline. Only the resulting THCA model needs to be loaded into the embedded processor for eventual execution.

based Executive's control sequencer to execute timed control programs expressed as THCA.

5.3 Executing THCA

Informally, a THCA execution cycle proceeds as follows. The control sequencer begins with a marked subset of the THCA's locations, an estimate of the current plant state from mode estimation, and a clock interpretation providing a value for each active clock variable. It then creates a set consisting of each marked location whose maintenance constraint is satisfied by the current estimated state and clock interpretation. Next, it conjoins all goal constraints of this set to produce a *configuration goal.* A configuration goal represents a set of states; the plant must progress towards the state in this set with greatest reward (called the *goal state*). [2] This configuration goal is then passed to mode reconfiguration, which executes a single command that makes progress towards achieving the goal. Next, the sequencer receives an update of the plant state from mode estimation. Based on this new state information, the THCA advances to a new marking, by taking all enabled transitions from (a) marked primitive locations whose goal constraints are achieved, (b) marked primitive locations whose maintenance constraints have been violated, and (c) marked composite locations that no longer contain any marked subautomata. This execution model builds upon the execution of HCA [90], by handling clock initializations, checking for satisfaction of clock constraints in the maintenance constraints and transition guards, and updating clock variables based on the current time from the system clock.

More precisely, to execute a THCA $\mathcal{A}$, the control sequencer starts with an estimate of the initial state of the plant, $\hat{s}^{(0)}$, and an empty clock interpretation, $\omega^{(0)} = \emptyset$. It initializes $\mathcal{A}$ using $m_F(\mathcal{A})$, a function that marks the start locations of $\mathcal{A}$ and all their starting subautomata. As described in Chapter 4, execution proceeds in cycles. The length of an execution cycle is determined by the amount of time required for

[2] "Progress" is defined as taking an action that is part of a simple sequence of actions that leads to the goal state.

the Timed Model-based Executive to complete its control sequencer and deductive controller operations (further detail on the order of operations in an execution cycle is deferred to Chapter 7). In each execution cycle, the control sequencer steps automaton $\mathcal{A}$ using the function $Step_{THCA}$, which maps the current state estimate, clock interpretation and marking to a next state estimate, clock interpretation, marking and configuration goal. The functions m_F and $Step_{THCA}$ are defined below. Execution completes when no marks remain, since the empty marking is a fixed point.

Recall from Section 4.2 that a timed control program's timing mechanism is implemented via a set of active clock variables x_j^t, similar to the clock variables defined in Timed Automata [2]. In any execution cycle, an assignment to an active clock variable represents the amount of metric time elapsed since that clock variable was initialized. Clock variable assignments are updated at the beginning of each execution cycle, and remain constant throughout. The clock interpretation $\omega^{(i)}$ consists of the set of assignments to all active clock variables at the start of each execution cycle i. The assignment to each active clock variable x_j^t is computed as the difference between the absolute system time at the start of cycle i, $t^{abs}(i)$, and the absolute system time when clock j was initialized, $t_j^{init}(i)$. Unlike the clocks used in Timed Automata, active clocks are never "reset" to zero in a THCA, as discussed in Section 4.2.

Given a THCA $\mathcal{A}$ to be initialized, $m_F(\mathcal{A})$ creates a *full marking*, by recursively marking the start locations of $\mathcal{A}$ and all starting subautomata of these start locations:

$$\begin{aligned} m_F(\mathcal{A}) \;=\; & \mathcal{A} \cup \bigcup \{\sigma^p_{start} \mid \sigma^p_{start} \in \Sigma_p(\mathcal{A}) \cap \Theta(\mathcal{A})\} \\ & \cup \bigcup \{m_F(\sigma^c_{start}) \mid \sigma^c_{start} \in \Sigma_c(\mathcal{A}) \cap \Theta(\mathcal{A})\}. \end{aligned}$$

$Step_{THCA}$ transitions an automaton $\mathcal{A}$ from the current full marking to the next full marking and generates a new configuration goal, based on the current state estimate and clock values. The $Step_{THCA}$ algorithm is given in Figure 5-3. Key features of the algorithm are as follows. *Reactive preemption* is implemented in step 2, which unmarks all subautomata of composite locations whose maintenance constraints have been violated by the latest state estimate, preventing the assertion of

any goal constraints by these subautomata. *Clock persistence* is ensured in step 3, where an already-active clock does not get reset to zero if the primitive location gets re-marked, for example due to incomplete achievement of its goal constraint. *Goal-driven execution* is provided by steps 4 and 5 via assertion of the configuration goal for achievement by the deductive controller. *Closed-loop execution* is implemented in steps 6 and 7, based on state feedback from the plant; a goal continues to be asserted until it is determined to have been achieved by the deductive controller. Finally, *progress due to goal achievement or preemption* is ensured in step 8; by enabling transitions out of locations whose maintenance constraints have been violated, the language can encode sequenced reactions to preemption events, in addition to traditional catch/throw mechanisms.

A *trajectory* of a THCA $\mathcal{A}$, given estimated plant state sequence $[\hat{s}^{(0)}, \hat{s}^{(1)}, \ldots]$ and absolute system time sequence $[t^{abs}(0), t^{abs}(1), \ldots]$, is a sequence of markings $[m^{(0)}, m^{(1)}, \ldots]$, configuration goals $[g^{(0)}, g^{(1)}, \ldots]$, and clock initialization times $[t^{init}(0), t^{init}(1), \ldots]$, such that: (a) $m^{(0)}$ is the initial marking $m_F(\mathcal{A})$; (b) $t^{init}(0)$ is an empty set; and (c) for each $i \geq 0$, $\langle g^{(i)}, m^{(i+1)}, \hat{s}^{(i+1)}, t^{init}(i+1)\rangle = Step_{THCA}(\mathcal{A}, m^{(i)}, \hat{s}^{(i)}, t^{init}(i), t^{abs}(i))$. *$\mathcal{A}$'s execution completes at step i_f* if $m^{(i_f)}$ is the empty marking, and there is no $i < i_f$ such that $m^{(i)}$ is the empty marking.

5.4 THCA Execution Example

The tight interaction between the control sequencer and the mode estimation and mode reconfiguration capabilities of the deductive controller is best illustrated by walking through a simple example. Consider a nominal (i.e., failure-free) execution trace for the Mars entry sequence introduced in Chapter 3. Markings are represented in Figures 5-4 to 5-12 by filling in the corresponding primitive locations. Any composite location with marked subautomata is considered marked. Locations are numbered 1-12 in the figures, for reference.

Initial State – Initially, all start locations are marked (locations 1 and 2 in Figure 5-4). Mode estimation provides the following initial plant state estimate:

$Step_{THCA}(\mathcal{A}, m^{(i)}, \hat{s}^{(i)}, t^{init}(i), t^{abs}) \rightarrow \langle g^{(i)}, m^{(i+1)}, \hat{s}^{(i+1)}, t^{init}(i+1) \rangle ::$

1. **Update active clocks.** Clock variable assignments in $\omega^{(i)}$ are computed from t^{abs} and $t^{init}(i)$. Update each active clock variable ($x_j^t = t^{abs} - t_j^{init}$). Add these new clock variable assignments to $\omega^{(i)}$.

2. **Check maintenance constraints for marked composites.** Unmark all subautomata of any marked composite location in $m^{(i)}$ whose maintenance constraint is violated by $\hat{s}^{(i)} \wedge \omega^{(i)}$.

3. **Assert clock initializations.** For any clock initialization ($x_j^t = 0$) asserted by currently marked primitive locations, unless t_j^{init} is already specified in $t^{init}(i)$, set $t_j^{init}(i+1) = t^{abs}$.

4. **Setup configuration goal.** Output, as the configuration goal $g^{(i)}$, the conjunction of goal constraints from currently marked primitive locations in $m^{(i)}$ whose maintenance constraints are satisfied by $\hat{s}^{(i)} \wedge \omega^{(i)}$.

5. **Take action.** Request that mode reconfiguration issue a command that progresses the plant towards a state that achieves the configuration goal $g^{(i)}$.

6. **Read next state estimate.** Once the command has been issued, obtain from mode estimation the plant's new most likely state $\hat{s}^{(i+1)}$.

7. **Await incomplete goals.** If the goal constraint of a primitive location marked in $m^{(i)}$ is not entailed by $\hat{s}^{(i+1)}$, and its maintenance constraint was not violated by $\hat{s}^{(i)} \wedge \omega^{(i)}$, then include that location as marked in $m^{(i+1)}$.

8. **Identify enabled transitions.** A transition from a marked primitive location σ^p in $m^{(i)}$ is enabled if both of the following conditions hold true:

   (a) σ^p's goal constraint is entailed by $\hat{s}^{(i+1)}$, or its maintenance constraint was violated by $\hat{s}^{(i)} \wedge \omega^{(i)}$;

   (b) the transition's guard condition is satisfied by $\hat{s}^{(i+1)} \wedge \omega^{(i)}$.

   A transition from a marked composite location σ^c in $m^{(i)}$ is enabled if both of the following conditions hold true:

   (a) none of σ^c's subautomata are marked in $m^{(i+1)}$ and none of σ^c's subautomata have enabled outgoing transitions;

   (b) the transition's guard condition is satisfied by $\hat{s}^{(i+1)} \wedge \omega^{(i)}$.

9. **Take transitions.** Mark and initialize in $m^{(i+1)}$ the target of each enabled transition, using function m_F. Re-mark in $m^{(i+1)}$ all composite locations with subautomata that are marked in $m^{(i+1)}$.

10. **Update list of clock references.** Add the contents of $t^{init}(i)$ to $t^{init}(i+1)$.

Figure 5-3: $Step_{THCA}$ algorithm.

{*Engine=Off, Nav=Earth-relative, Att=Cruise-orient, Lander=Connected, Entry=Not-initiated*}.

Execution will continue as long as outermost composite location 1 remains marked. Note that system time values associated with each cycle are for illustrative purposes only, and do not reflect the actual rate of execution of the control sequencer.

Cycle 1, system time $t^{abs} = 0.0$ sec – The first $Step_{THCA}$ cycle proceeds as follows. (Step 1) There are no active clocks, so no clock variables are updated. (Step 2) Since no maintenance constraints are specified for marked composite location 1,

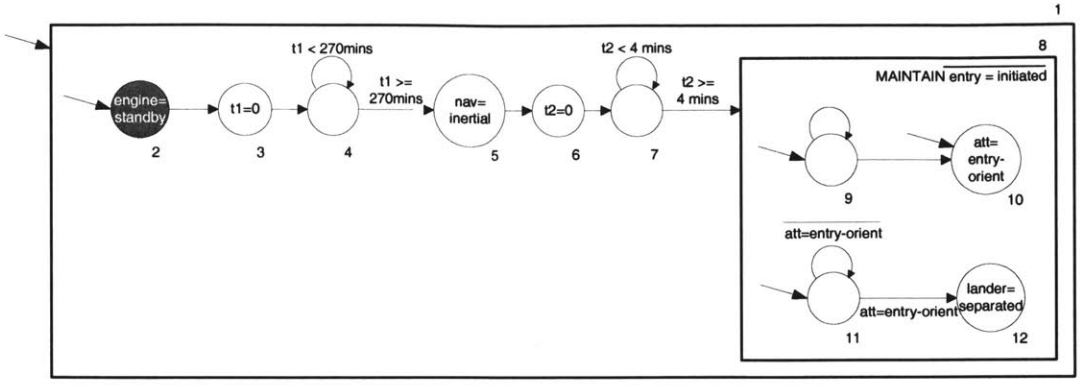

Figure 5-4: Initial marking of the THCA for Mars entry, corresponding to Cycles 1 to N_1.

it remains marked. (Step 3) No clocks are initialized by the start locations. (Step 4) The only goal constraint asserted is *Engine=Standby*. This state assignment is passed to MR as the configuration goal. (Step 5) MR issues the first command in a sequence that achieves the configuration goal. Based on the simple plant model shown in Figure 3-3, this command is *cmd=standby*. (Step 6) ME returns the new state estimate, which indicates that *Engine=Heating* is achieved as a result of this command. (Step 7) The goal constraint for the only marked primitive location has not yet been achieved, so location 2 will remain marked in the next cycle. (Step 8) Since location 2's goal constraint is not yet achieved, its outgoing transition is not enabled. (Step 9) Since no transitions are enabled, locations 1 and 2 remain marked for the next execution cycle. Thus the next execution step's marking remains as shown in Figure 5-4. (Step 10) There are no active clocks, so no initialization times to update.

Cycle 2, system time t^{abs} = 0.5 sec – Skipping the uninteresting algorithm steps, the next $Step_{THCA}$ cycle proceeds as follows. (Steps 4-5) The configuration goal *Engine=Standby* is still being asserted by location 2. MR reasons through the plant model and sees that the engine's standby mode is reachable without taking any action, so it returns no command. (Step 6) ME returns the new state estimate, which is the same as the previous estimate. (Step 7) The goal constraint for location 2 has not yet been achieved, so it will remain marked in the next cycle. (Steps 8-9) Thus,

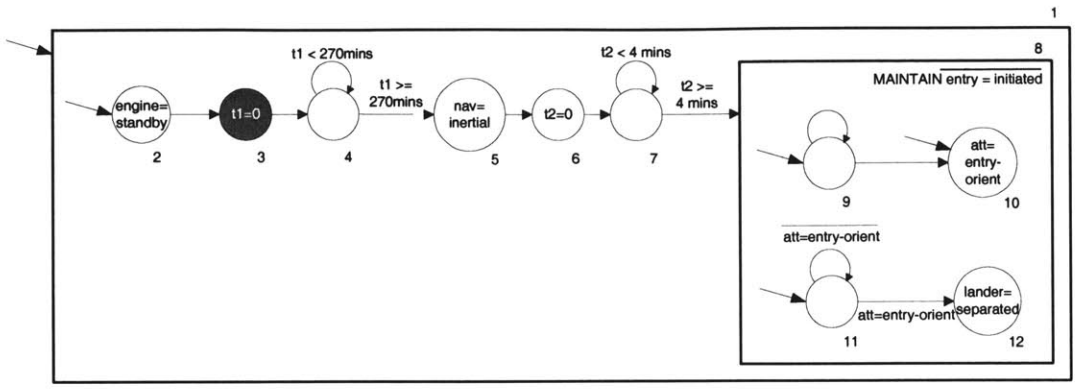

Figure 5-5: Marking of the Mars Entry THCA for Cycle $(N_1 + 1)$.

its outgoing transition is not enabled, so locations 1 and 2 remain marked for the next cycle (Figure 5-4).

Since, according to the component model in Figure 3-3, it takes between 30 and 60 seconds for the engine to reach standby mode, the execution behavior from Cycle 2 will be repeated for multiple cycles. Consequently, this discussion skips to the next "interesting" execution cycle, Cycle N_1, which occurs 43.2 seconds later. The state estimate {*Engine=Heating, Nav=Earth-relative, Att=Cruise-orient, Lander=Connected, Entry=Not-initiated*} is unchanged, and locations 1 and 2 remain marked from Cycle 3 to Cycle N_1.

Cycle N_1, system time t^{abs} = 43.2 sec – (Steps 4-5) The goal constraint *Engine=Standby* is asserted as the configuration goal. MR again returns no command. (Step 6) ME determines from the latest observations that the engine has reached standby mode. (Step 7) Since location 2's goal constraint is now achieved, it will not remain marked in the next cycle. (Step 8) Since location 2's outgoing transition is labeled **True**, it is enabled. (Step 9) After taking the enabled transition, the marked locations are 1 and 3, as shown in Figure 5-5.

Cycle $(N_1 + 1)$, system time t^{abs} = 43.8 sec – (Step 3) Location 3 asserts the clock initialization *t1=0*, so the current system time of 43.8 sec is stored as the initialization time for *t1*. (Steps 4-5) No goal constraints are asserted by the set of marked locations, so no action needs to be taken by MR. (Step 6) ME's new state estimate remains unchanged from the previous cycle. (Step 8) Since location 3's

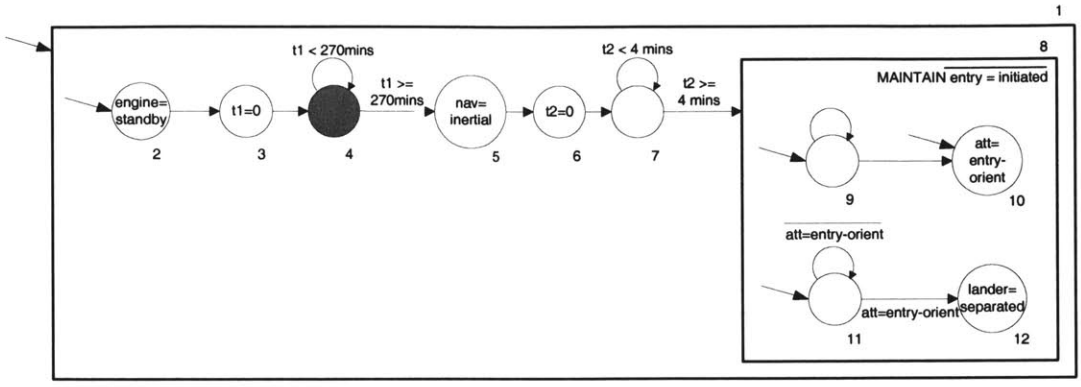

Figure 5-6: Marking of the Mars Entry THCA for Cycles $(N_1 + 2)$ to N_2.

outgoing transition is labeled **True**, it is enabled. (Step 9) After taking the enabled transition, the marked locations are 1 and 4, as shown in Figure 5-6. (Step 10) $t_1^{init} = 43.8$ sec is added to the set of initialization times for the next cycle.

Cycle (N_1+2), system time t^{abs} = 44.2 sec – (Step 1) Clock variable *t1 = 44.2 - 43.8 = 0.4 sec* is updated. (Steps 3-6) No clock initializations or goal constraints are asserted by the marked primitive location 4, and ME's state estimate remains unchanged from the previous cycle. (Step 8) Since the value of *t1* is less than 270 minutes, only the transition from location 4 to itself is enabled. (Step 9) After taking the enabled transition, locations 1 and 4 remain marked (Figure 5-6).

Since this behavior will be repeated for multiple cycles, the discussion again skips to the next "interesting" execution cycle, Cycle N_2, which occurs 16199.8 seconds later. The state estimate {*Engine=Standby, Nav=Earth-relative, Att=Cruise-orient, Lander=Connected, Entry=Not-initiated*} is unchanged, and locations 1 and 4 remain marked from Cycle $(N_1 + 3)$ to Cycle (N_2).

Cycle N_2, system time t^{abs} = 16244.0 sec – Clock variable *t1 = 16244.0 - 43.8 = 16200.2 sec* is updated. No clock initializations or goal constraints are asserted and the new state estimate remains unchanged. Since the value of *t1* is greater than 270 minutes, only the transition from location 4 to location 5 is enabled. After taking the enabled transition, locations 1 and 5 are marked (Figure 5-7).

Cycle (N_2+1), system time t^{abs} = 16244.5 sec – Clock variable *t1 = 16244.5 - 43.8 = 16200.7 sec* is updated. Location 5 asserts the goal constraint *Nav=Inertial.*

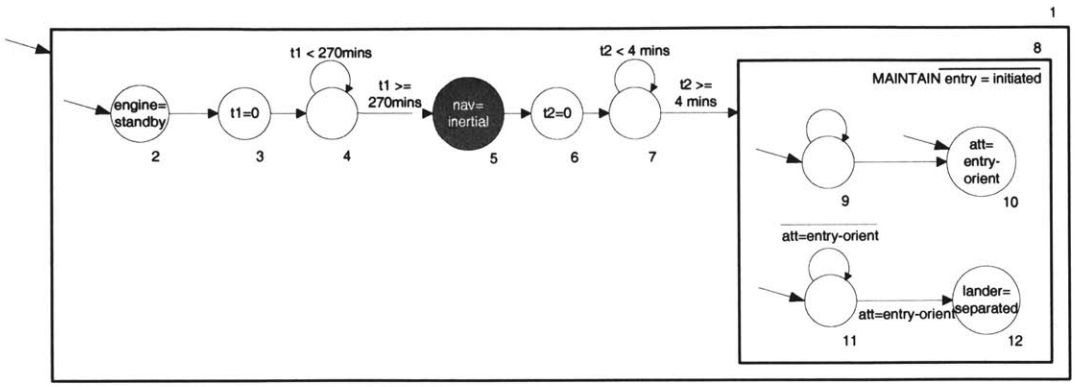

Figure 5-7: Marking of the Mars Entry THCA for Cycle $(N_2 + 1)$.

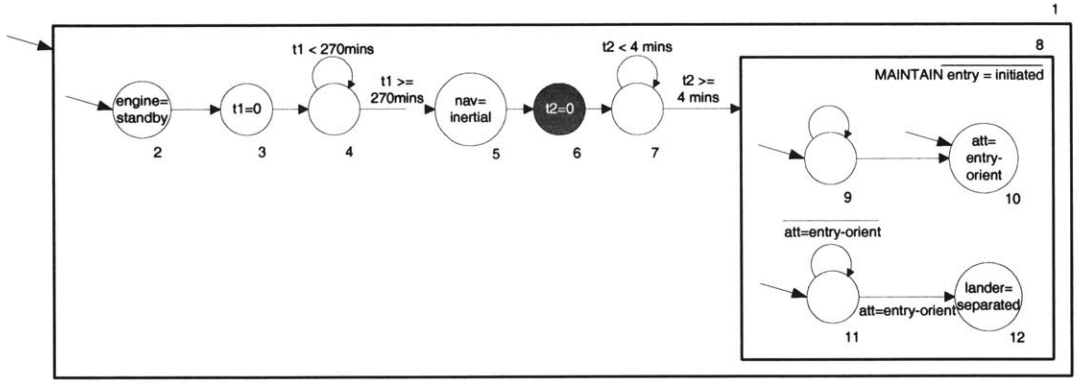

Figure 5-8: Marking of the Mars Entry THCA for Cycle $(N_2 + 2)$.

This state assignment is passed to MR as the configuration goal. MR issues the first command in a sequence that achieves the configuration goal. ME confirms that *Nav=Inertial* is achieved as a result of this command. The goal constraint for the only marked primitive location has been achieved, so it will not remain marked in the next cycle. Since location 5's outgoing transition is labeled **True**, it is enabled. After taking the enabled transition, the marked locations are 1 and 6, as shown in Figure 5-8.

Cycle (N_2+2), system time t^{abs} = 16245.1 sec – Clock variable *t1 = 16245.1 - 43.8 = 16201.3 sec* is updated. Location 6 asserts the clock initialization *t2=0*, so the current system time of 16245.1 sec is stored as the initialization time for *t2*. ME's new state estimate remains unchanged from the previous cycle. Since location 6's outgoing transition is labeled **True**, it is enabled. After taking the enabled transition,

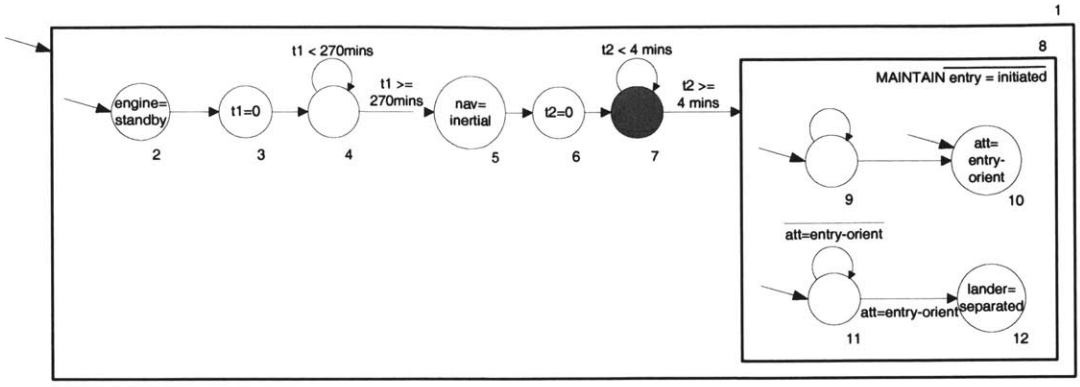

Figure 5-9: Marking of the Mars Entry THCA for Cycles $(N_2 + 3)$ to N_3.

the marked locations are 1 and 7 (Figure 5-9). t_2^{init} = 16245.1 sec is added to the set of initialization times for the next cycle.

Cycle $(N_2 + 3)$, system time t^{abs} = 16245.6 sec – Both clock variables *t1* = *16245.6 - 43.8 = 16201.8 sec* and *t2 = 16245.6 - 16245.1 = 0.5 sec* are updated. Since the value of *t2* is less than 4 minutes, only the transition from location 7 to itself is enabled. Thus, in the next cycle, locations 1 and 7 remain marked (Figure 5-9).

Again skipping to the next "interesting" execution cycle, Cycle N_3, which occurs 239.9 seconds later, the state estimate {*Engine=Standby, Nav=Inertial, Att=Cruise-orient, Lander=Connected, Entry=Not-initiated*} is unchanged. Locations 1 and 7 remain marked from Cycle $(N_2 + 3)$ to Cycle (N_3).

Cycle N_3, system time t^{abs} = 16485.5 sec – Clocks *t1 = 16485.5 - 43.8 = 16441.7 sec* and *t2 = 16485.5 - 16245.1 = 240.4 sec* are updated. Since the value of *t2* is greater than 4 minutes, only the transition from location 7 to composite location 8 is enabled. After taking the enabled transition and marking location 8's starting subautomata, the marked locations for the next cycle are 1, 8, 9, 10, and 11, as shown in Figure 5-10.

Cycle (N_3+1), system time t^{abs} = 16486.0 sec – The maintenance constraint corresponding to non-entailment of *Entry=Initiated* on location 8 holds for the current state estimate, so all its subautomata remain marked. The only goal constraint asserted is *Att=Entry-orient.* MR issues the first command in a sequence that achieves this configuration goal. The following ME update indicates that *Att=Slew-to-Entry,*

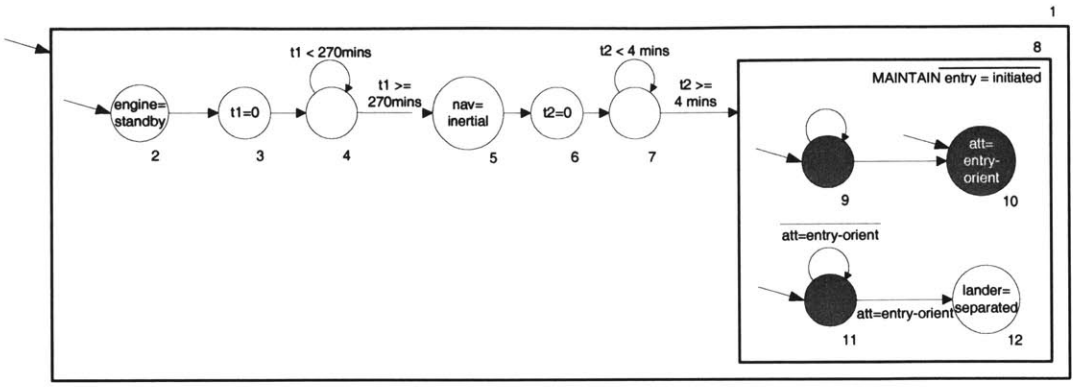

Figure 5-10: Marking of the Mars Entry THCA for Cycles $(N_3 + 1)$ to N_4.

so *Att=Entry-orient* is not yet achieved. Consequently, location 10 remains marked in the next cycle. The transitions from locations 9 to 9, 9 to 10, and 11 to 11 are enabled by the current state. After taking these transitions, locations 1, 8, 9, 10 and 11 remain marked, as in Figure 5-10.

It takes 12 seconds for the spacecraft to complete its rotation to the entry orientation, as a result of the closed-loop goal-driven commanding of the executive; the behavior from Cycle $(N_3 + 1)$ will repeat over multiple execution cycles. Discussion skips once again to the next "interesting" cycle, Cycle N_4, which occurs 12.2 seconds later.

Cycle N_4, system time t^{abs} = 16498.2 sec – The maintenance constraint on location 8 still holds, and the only goal constraint asserted is *Att=Entry-orient.* This time, ME confirms achievement of this goal. Taking the enabled transitions leads to a new set of marked locations: 1, 8, 9, 10 and 12 (Figure 5-11). Note that location 10 remains marked despite achievement of its goal constraint, due to re-marking by the transition from 9 to 10. This reflects a desire to hold the spacecraft at the entry orientation.

Cycle $(N_4 + 1)$, system time t^{abs} = 16498.8 sec – The maintenance constraint on location 8 still holds. The configuration goal passed to MR now consists of *Att=Entry-orient* and *Lander=Separated.* The following ME update confirms achievement of both goal states. Since location 12 has no outgoing transitions, its thread of execution dies. Thus, taking the enabled transitions leads to marked locations 1, 8,

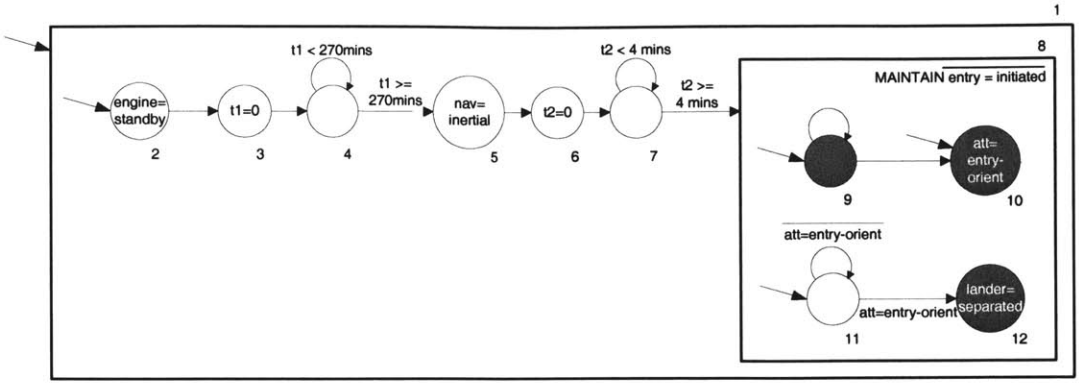

Figure 5-11: Marking of the Mars Entry THCA for Cycles $(N_4 + 1)$ to N_5.

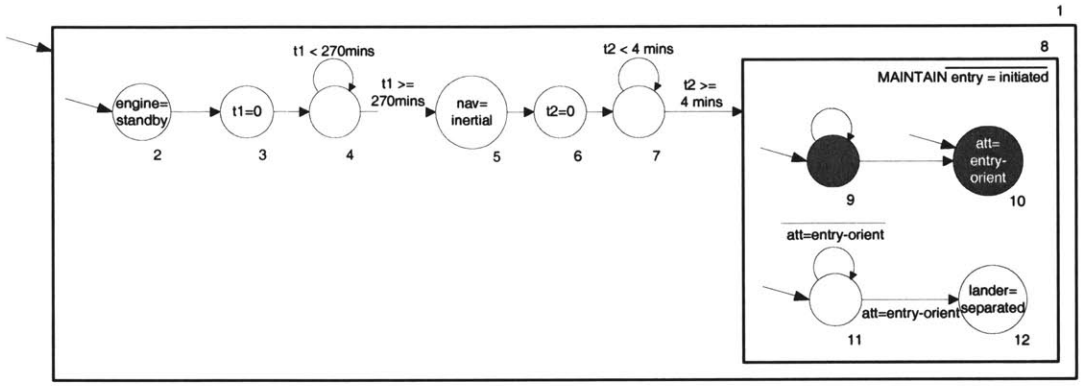

Figure 5-12: Marking of the Mars Entry THCA from Cycle $(N_4 + 1)$ to the cycle when ME determines that *Entry=Initiated.*

9, and 10, as in Figure 5-12.

Remaining Cycles – Execution continues with locations 1, 8, 9, and 10 remaining marked (and *Att=Entry-orient* continuing to be asserted as a goal) until ME indicates that *Entry=Initiated.* At this point, location 8's maintenance constraint becomes violated. It has no outgoing transitions, so its thread of execution dies. Since none of location 1's subautomata remain marked, it also becomes unmarked, and execution of the Mars Entry THCA terminates.

5.5 Summary

In this chapter, the RMPL and THCA languages for encoding timed control programs have been formally specified. THCA provide a compact, hierarchical implementation of the semantic model for the timed control program presented in Section 4.2. The THCA language extends the HCA representation from model-based programming [90] by incorporating various notions from real-time modeling formalisms like Timed Automata [2], including clock variables, clock interpretations, clock initializations, and timing constraints. THCA is distinguished from these other real-time modeling formalisms by its adoption of a hierarchical computational model and its use of constraints on plant state, in the form of goal constraints and maintenance constraints.

The execution algorithm for THCA has also been presented. In each execution cycle, the control sequencer steps a THCA by mapping the current state estimate, clock interpretation and program marking to a next state estimate, clock interpretation, program marking and configuration goal. Key features of the algorithm include reactive preemption, clock persistence, goal-driven and closed-loop execution, and progress due to goal achievement or preemption. The tight interaction between the control sequencer and the deductive controller has been illustrated by walking through the execution of the Mars entry control program.

Chapter 6

Deductive Controller

In this chapter, the deductive controller for the Timed Model-based Executive is described. The deductive controller uses a timed plant model to estimate and control the state of the system. Recall that a physical plant is comprised of discrete and analog hardware and software. As discussed in Section 4.1, the behavior of the plant is modeled as a factored POSMDP, which is encoded compactly using probabilistic *Timed Concurrent Constraint Automata (TCCA)*. Constraints are used to represent cotemporal interactions between state variables and intercommunication between components. Probabilistic timed transitions are used to model the stochastic behavior of components, such as failure and latency. Reward is used to assess the costs and benefits associated with particular component modes.

This chapter begins by defining TCCA as a composition of *Timed Constraint Automata* for individual components of a plant. The TCCA encoding of the plant model is essential to the efficient operation of the deductive controller (see Figure 6-1). The two core capabilities of the deductive controller, ME and MR, are then developed. These two capabilities reason through timed plant models encoded as TCCA, to estimate the plant state (ME) and generate an appropriate control sequence that achieves a configuration goal (MR).

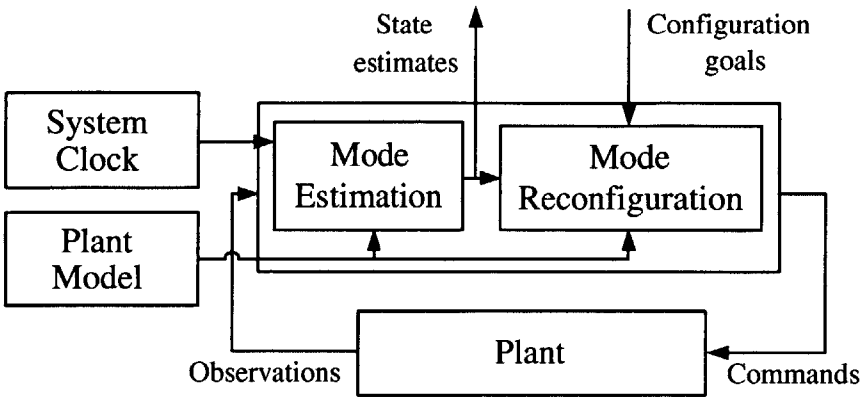

Figure 6-1: Architecture of the deductive controller.

6.1 Plant Models as Timed Concurrent Constraint Automata

In this section, components of the physical plant are first defined in terms of probabilistic transition systems with time bounds associated with transitions between modes. These systems are then re-cast as *Timed Constraint Automata*, by mapping each time-bounded transition to an intermediate mode representing the state of going from the source mode to the target mode of the transition.

The full plant model is represented in terms of *Timed Concurrent Constraint Automata (TCCA)*. TCCA are defined as a composition of Timed Constraint Automata for the individual components. The component automata operate concurrently, synchronized with a global system clock. Prior model-based executives considered untimed plant models (i.e., without time bounds on the transitions). For these executives, the nominal trajectories of the plant were defined as deterministic executions of (untimed) Concurrent Constraint Automata [83, 85, 90]. Now that the plant model includes time-bounded transitions, even a nominal (fault-free) execution must be defined in terms of non-deterministic executions of TCCA. In particular, specifying a range of time that it takes for a component to transition introduces non-determinism into the nominal model. Probabilistic transitions to fault modes provide another source of non-determinism in the model.

6.1.1 Physical Plant Component Modeling

The behavior of each component of the physical plant can be captured in a transition system, as shown in Figure 6-2. Each component is represented by a set of component modes (represented as circles in the figure), a set of constraints defining the behavior within each mode (specified in boxes adjacent to each mode), and a set of time-bounded probabilistic transitions between modes (represented by the arrows in the figure).

Conceptually, a mode of a component can be thought of as a state in which distinct behavior is observed, and can be described in terms of a constraint on plant variables with finite domains. Components can only be in a single mode at a time. Component modes are specified for both nominal and off-nominal behavior. Transitions with nominal modes as their target are defined as *nominal transitions*, and transitions with off-nominal (or *fault*) modes as their target are defined as *failure transitions*. Transitions are labeled with guard conditions on plant variables that must hold for the transition to be taken, time bounds corresponding to the upper and lower limits on the amount of time it can take the system to fully transition from the source to the target mode (once the corresponding guard has been satisfied), and probabilities corresponding to the likelihood that the transition will be taken when its guard condition is satisfied.

6.1.2 Timed Constraint Automata

In the above transition system representation, transitions between component modes take finite time, as specified in the associated time bounds. However, in traditional automata model formulations, transitions correspond to instantaneous changes in state. The component models can therefore be re-cast as Timed Constraint Automata, by mapping each transition with a time bound to an intermediate mode representing the transitional state of going from the source mode to the target mode of the transition (see Figure 6-3).

In the Timed Constraint Automaton formulation, the source and target mode

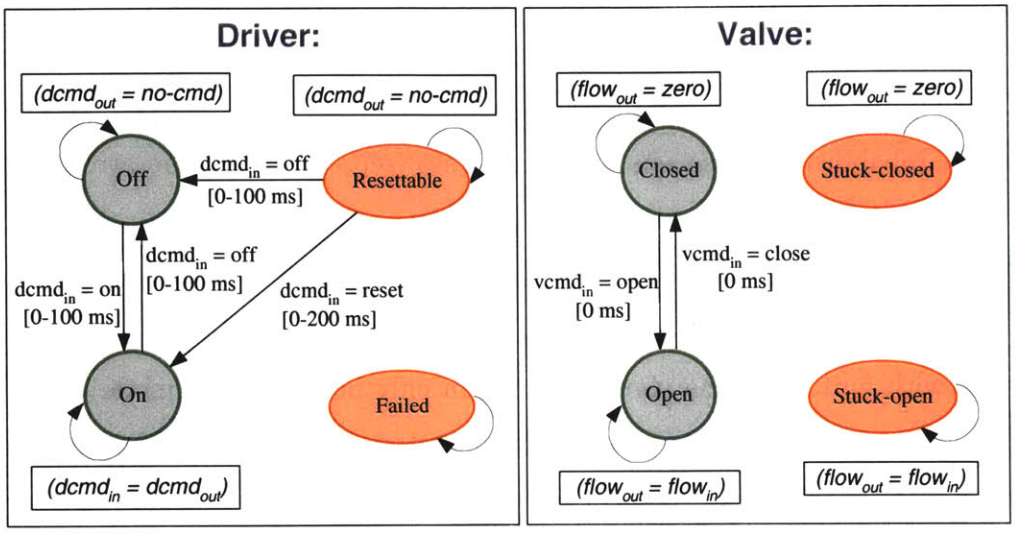

Figure 6-2: Component models for a driver and valve. Probabilities and fault transitions are omitted for clarity.

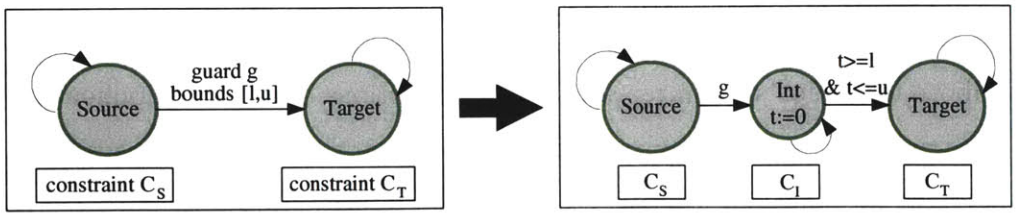

Figure 6-3: Mapping of a transition with time bounds to an intermediate (transitional) mode.

constraints remain the same as in the original transition system representation (i.e. C_S and C_T, respectively). Choice of an appropriate constraint C_I for the intermediate mode depends on the physics of the plant being modeled. Four possible approaches are discussed here:

1. If the transition from the source mode's behavior regime to the target mode's behavior regime occurs very quickly, but is simply delayed from its triggering event by time δ (with $l \leq \delta \leq u$), then an appropriate model would be $C_I = C_S$. Since the component behavior during the intermediate mode representing the delay is the same as the source mode behavior, the constraints for these modes should be the same.

2. If the transition follows a slow, monotonic transient from the source mode's behavior regime directly into the target mode's behavior regime, then an appropriate model would be $C_I = C_S \vee C_T$. This allows for the behavior to transition smoothly from the source mode's behavior to the target mode's behavior, over the course of the sojourn in the intermediate mode.

3. If the transitional behavior is known *a priori* (e.g., as a result of empirical tests performed on the component), and can be explicitly captured by the modeler in a constraint (which might be completely unrelated to C_S and C_T), then this constraint is the best choice for C_I.

4. On the other hand, if the transitional behavior follows an arbitrary unknown transient prior to stabilization into the target mode, then it would be most appropriate to model the behavior while in the intermediate mode as *unconstrained*, i.e. C_I is specified by the empty constraint **True**. This captures the most general behavior during the transition.

In this thesis, approach #3 is adopted, where the specific transitional behavior is generally available to the modeler.

In the mapping shown in Figure 6-3, the transition from the source mode to the intermediate mode is conditioned on the original transition's guard constraint

(represented as label g, corresponding to a propositional logic constraint on system variable assignments). Upon transitioning into the intermediate mode, a plant clock is started (represented by the assertion $t := 0$ in the figure). The transition from the intermediate mode to the target mode is conditioned on a time constraint on the clock variable, represented as label $(t \geq l) \wedge (t \leq u)$ in the figure. Thus, all transitions between modes in a Timed Constraint Automaton are assumed to be instantaneous, and conditioned on system variable assignments or clock variable constraints. Note that timed transitions are taken at some time δ that satisfies the time constraint, but not necessarily at the first time that does so (i.e. a transition with the time constraint $(t \geq l) \wedge (t \leq u)$ is not necessarily taken at time $t = l$). The non-deterministic nature of timed transitions differentiates Timed Constraint Automata from the Constraint Automata defined in [90].

The model of a physical plant is composed of a set of Timed Constraint Automata, one for each component in the model (e.g., Figure 6-4). Each Timed Constraint Automaton has an associated mode variable x_j with domain $\mathbb{D}(x_j)$. $\mathbb{D}(x_j)$ is partitioned into a set of nominal modes and a set of fault modes. A component's behavior within each mode is modeled by an associated constraint $\mathbb{M}_j(x_j = v)$ over the system variables. Each mode also has an associated reward $\mathbb{R}_j(x_j = v)$. Given a current mode assignment, $x_j = v$, the component changes its mode by taking an *enabled* outgoing transition τ. A transition is enabled if its guard condition and time constraint are satisfied. A probability $\mathbf{P}_\tau(t_j)$ is associated with each transition τ. $\mathbf{P}_\tau(t_j)$ is the probability that τ is taken when enabled, for a given value of the clock variable t_j. For a transition τ with a time constraint, this can be viewed as associating with τ a probability density function $\mathbf{p}_\tau$, over the possible values of the clock variable t_j. For example, given a transition conditioned on a time constraint of the form $(t_j \geq l) \wedge (t_j \leq u)$, $\mathbf{p}_\tau$ should equal zero for all values of t_j up to the lower bound and greater than the upper bound, and should integrate to one from the lower bound to the upper bound. This probability density function is specified by the modeler, based on engineering knowledge of the component's stochastic behavior (e.g., as derived from analysis of the design specification of the component, or determined from

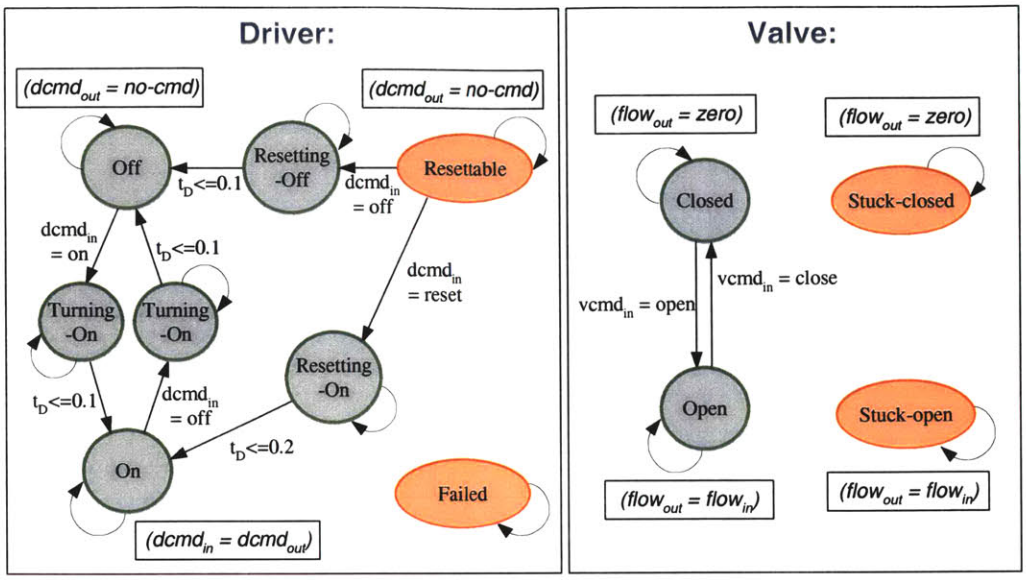

Figure 6-4: Timed Constraint Automata for the driver and valve components. The names of the driver's four transitional modes are *TurningOn* (from *Off* to *On*), *TurningOff* (from *On* to *Off*), *ResettingOn* (from *Resettable* to *On*), and *ResettingOff* (from *Resettable* to *Off*). Probabilities and fault transitions are omitted for clarity.

empirical testing).

Modeling physical systems in terms of Timed Constraint Automata

Though Timed Constraint Automata have been introduced in the context of a mapping from a probabilistic transition system with time-bounded transitions, it should be noted that these automata allow for more general modeling of physical behavior than the original probabilistic transition system. Consider the simple models in Figure 6-5, expressed directly in terms of Timed Constraint Automata. On the left, a simple spacecraft engine is modeled with four nominal modes: *Off*, *Heating*, *Standby*, and *Firing*. The *Heating* mode corresponds to a transitional state that captures the fact that it takes time for the engine to heat up to its nominal standby temperature. This heating process can take anywhere from 30 to 60 seconds to complete. However, this model also captures the following important feature: an engine in the midst of heating up can still be commanded off (preempting the heating-up process). If this model had been expressed in terms of the type of probabilistic transition system

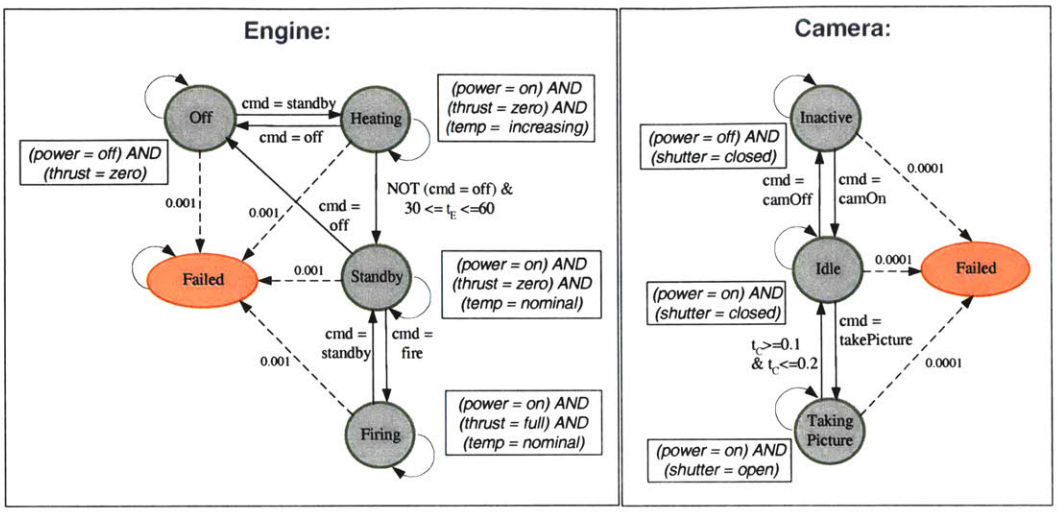

Figure 6-5: Timed Constraint Automata for the engine and camera components.

shown in Figure 6-2, the resulting mapping to a Timed Constraint Automaton would not have produced this preempting transition.

On the right in Figure 6-5, a simple camera component is modeled. When the "takePicture" command is issued from the *Idle* mode, the following behavior is observed: the camera shutter opens for between 100 and 200 milliseconds, then closes automatically. This behavior is captured as a transient *Taking-Picture* mode, with an outgoing timed transition back to *Idle* mode. Again, this type of transitional behavior cannot be accurately represented by the form of transition system in Figure 6-2. Consequently, physical systems are generally modeled directly in the Timed Constraint Automaton representation, to take advantage of the greater expressivity it provides.

Formal definition of Timed Constraint Automata

The Timed Constraint Automaton model for component j is described by a tuple $\langle \Pi_j, \mathbb{M}_j, \mathbb{I}_j, \mathbb{T}_j, \mathbf{P}_{\Theta_j}, \mathbf{P}_{\mathbb{T}_j}, \mathbb{R}_j \rangle$, where:

- Π_j is a set of variables, partitioned into Π_j^m, Π_j^t and Π_j^a:
  - Π_j^m is the singleton set containing the component's mode variable x_j. x_j ranges over the finite domain $\mathbb{D}(x_j)$.

- Π_j^t is the singleton set containing the component's clock variable t_j. t_j ranges over $\mathbb{D}(t_j) = \Re^+$. $\mathbb{C}_{time}[\Pi_j^t]$ is the set of all possible clock variable constraints over t_j of the form $\lambda \triangleq TRUE \mid FALSE \mid (t_j \leq t_1) \mid (t_j \geq t_1) \mid (t_j < t_1) \mid (t_j > t_1) \mid \lambda_1 \wedge \lambda_2 \mid \lambda_1 \vee \lambda_2$, where $t_1 \in \Re^+$. $\mathbb{C}_{init}[\Pi_j^t]$ is the singleton set $\{(t_j := 0)\}$, where $(t_j := 0)$ is the clock initialization of t_j.

- Π_j^a is the set of *attribute variables* a_j. Attributes include input variables, output variables, and any other variables needed to describe the modeled behavior of the component. Each $a_j \in \Pi_j^a$ ranges over a finite domain $\mathbb{D}(a_j)$. $\mathbb{C}[\Pi_j^a]$ denotes the set of all finite-domain constraints over variables in Π_j^a of the form $\lambda \triangleq TRUE \mid FALSE \mid (a_j = v) \mid \neg\lambda_1 \mid \lambda_1 \wedge \lambda_2 \mid \lambda_1 \vee \lambda_2$, where $v \in \mathbb{D}[a_j]$.

- $\mathbb{M}_j : \mathbb{D}(x_j) \rightarrow \mathbb{C}[\Pi_j^a]$ associates with each mode variable assignment $x_j = v$ a finite domain constraint $\mathbb{M}_j(x_j = v) \in \mathbb{C}[\Pi_j^a]$. This constraint captures the component's behavior in a given mode.

- $\mathbb{I}_j : \mathbb{D}(x_j) \rightarrow \mathbb{C}_{init}[\Pi_j^t]$ associates with each mode variable assignment $x_j = v$ a clock initialization $(t_j := 0)$ that is asserted upon transition into this mode from a different mode.

- $\mathbb{T}_j \subseteq \mathbb{D}(x_j) \times \mathbb{C}[\Pi_j^a] \times \mathbb{D}(x_j)$ is the set of transitions. Each transition $\tau = \langle v, g, v' \rangle \in \mathbb{T}_j$ has a source mode $(x_j = v)$, a target mode $(x_j = v')$, and is conditioned on the guard constraint $g \in \mathbb{C}[\Pi_j^a]$. Transitions with nominal modes as their target are called *nominal* transitions; all others are called *fault* transitions.

- $\mathbf{P}_{\Theta_j} : \mathbb{D}(x_j) \rightarrow \Re[0, 1]$ denotes the probability that $x_j = v$ is the initial mode for component j.

- $\mathbf{P}_{\mathbb{T}_j} : \mathbb{T}_j \times \mathbb{D}(t_j) \rightarrow \Re[0, 1]$ associates, with each transition $\tau = \langle v, g, v' \rangle \in \mathbb{T}_j$ and value of clock variable t_j, a probability $\mathbf{P}_\tau(t_j)$. This corresponds to the probability that the transition will be taken if enabled at a given value of clock

variable t_j. $\mathbf{P}_{\mathbb{T}_j}$ can be viewed as associating with each timed transition a probability density function $\mathbf{p}_\tau(t_j)$, over the possible values of the clock variable t_j. This provides the mechanism for enforcing the time constraints on transitions, where each time constraint is an element in $\mathbb{C}_{time}[\Pi_j^t]$.

- $\mathbb{R}_j : \mathbb{D}(x_j) \rightarrow \Re$ denotes the cost (or conversely, the reward) associated with mode variable assignment $x_j = v$.

This definition of Timed Constraint Automata builds upon the definition of Constraint Automata for untimed plant models [83, 90]. In particular, Timed Constraint Automata extend Constraint Automata by defining the clock variables t_j and clock initializations $\mathbb{I}_j$, and by conditioning the transition probability functions $\mathbf{P}_{\mathbb{T}_j}$ on the clock variables.

These time-related augmentations lead to similarities between the Timed Constraint Automaton representation and two variants of Probabilistic Timed Automata (PTA) [52, 53]. PTA extend the Timed Automata formalism [2] by associating probabilities with the non-deterministic timed transitions. The PTA model proposed by Largouet and Cordier [53] assumes that the probability of taking a timed transition is constant for any time that satisfies the associated timing constraint. Timed Constraint Automata adopt a more realistic transition probability model, similar to the Continuous PTA model presented by Kwiatkowska et al. [52], by assuming the transition probability is computed based on the current clock value, according to a specified probability density function.

There are two key differences between Timed Constraint Automata and PTA. First, Timed Constraint Automata adopt a constraint-based encoding, where general propositional logic constraints on plant variables are used to specify behavior in each component mode ($\mathbb{M}_j$, in the above definition). PTA, on the other hand, simply associate with each node a set of "labels", i.e. atomic propositions that are true in that node. The only other constraints associated with nodes of PTA are the invariant clock conditions that must hold while in that state. Timed Constraint Automata do not explicitly define such invariant conditions on clocks; rather, these time constraints

are folded into the transition probability function $\mathbf{P}_{\mathbb{T}_j}$. Furthermore, transitions of Timed Constraint Automata are conditioned on general constraints on plant variable assignments, rather than event labels, as is the case for PTA.

The second key difference is that, unlike PTA, which allow for the specification of arbitrary numbers of clocks, each Timed Constraint Automaton defines a single clock. This restriction can be traced back to the adoption of a factored POSMDP model for physical plants: a single plant clock variable is all that is required to capture the types of semi-Markov behaviors exhibited in physical plant components, e.g. the latency associated with the heating of an engine component, or the non-deterministic duration of a camera's "taking-picture" action (see Figure 6-5). In this regard, the PTA provides a more general time-modeling capability, and thus can be used to describe systems with more complex timed behaviors. Nonetheless, Timed Constraint Automata are sufficiently expressive to model the rich set of timed physical behaviors described in the POSMDP framework.

6.1.3 Timed Concurrent Constraint Automata

A physical plant is modeled as a composition of Timed Constraint Automata that represent its individual concurrently operating components. This composition, including the interconnections between component automata and interconnections with the environment, is captured in a TCCA.

Formally, a TCCA is described by a tuple $\langle A, \Pi, \mathbb{Q} \rangle$, where:

- $A = \{A_1, A_2, ..., A_n\}$ denotes the finite set of Timed Constraint Automata associated with the n components in the plant.

- Π is a set of *plant variables*, partitioned into sets of *mode* variables $\Pi^m = \bigcup_{i=1..n} \Pi^m_j$, *control* variables $\Pi^c \subseteq \bigcup_{i=1..n} \Pi^a_j$, *observable* variables $\Pi^o \subseteq \bigcup_{i=1..n} \Pi^a_j$, *dependent* variables $\Pi^d \subseteq \bigcup_{i=1..n} \Pi^a_j$ and *clock* variables $\Pi^t = \bigcup_{i=1..n} \Pi^t_j$.

- $\mathbb{Q} \in \mathbb{C}(\Pi_c \cup \Pi_o \cup \Pi_d)$ is a conjunction of finite-domain constraints providing the interconnections between the attributes of the plant's components.

Mode variables represent the state of each component. Actuator commands are relayed to the plant through assignments to control variables. Observable variables capture the information provided by the plant's sensors. Dependent variables represent interconnections between components. They are used to transmit the effects of control actions and observations throughout the plant model. Finally, clock variables enable component transitions conditioned on metric time, allowing representation of delayed effects, for instance.

The state space of Π, denoted $\Sigma(\Pi)$, is the cross product of the $\mathbb{D}(var)$, for all variables $var \in \Pi$. The state space of the plant mode variables Π^m, denoted $\Sigma(\Pi^m)$, is the cross product of the $\mathbb{D}(x_j)$, for all variables $x_j \in \Pi^m$. A *state* of the plant in cycle i, $s^{(i)} \in \Sigma(\Pi^m)$, assigns to each component mode variable a value from its domain. Similarly, an *observation* of the plant, $o^{(i)} \in \Sigma(\Pi^o)$, assigns to each observable variable a value from its domain, and a *control action*, $\mu^{(i)} \in \Sigma(\Pi^c)$, assigns to each control variable a value from its domain. A *clock interpretation*, $\nu^{(i)} \in \Sigma(\Pi^t)$, assigns a real value to each clock variable in the system.

The above definition of TCCA as a composition of Timed Constraint Automata is analogous to the composition of untimed Constraint Automata into CCA [90]. Like CCA, TCCA model the evolution of physical processes by enabling and disabling constraints in a *constraint store*. Enabled constraints in the store would include the set of $\mathbb{M}_j(x_j = v)$ constraints imposed by the current plant state, and the $\mathbb{Q}$ constraints associated with the TCCA.

TCCA Specification Example

Consider the driver and valve component models depicted in Figure 6-4. The plant variables x_j associated with these components are *Driver* and *Valve*, with domains of {*TurningOn, On, TurningOff, Off, Resettable, ResettingOn, ResettingOff, Failed*} and {*Open, Closed, Stuck-open, Stuck-closed*}, respectively. The driver's constraint automaton has attributes dcmd_{in} and dcmd_{out}, each with domain {*on, off, reset, open, close, no-command*}. The valve's constraint automaton has attributes vcmd_{in} (with the same domain as dcmd_{in} and dcmd_{out}), flow_{in} (with domain {*zero, nominal*})

and flow_{out} (with the same domain as flow_{in}). The clock variables for the driver and valve are t_D and t_V, respectively.

For the driver component, the mode constraints $\mathbb{M}_j$ are specified as follows:

$$\mathbb{M}_{Driver}(\text{On}) = (\text{dcmd}_{in} = \text{dcmd}_{out})$$
$$\mathbb{M}_{Driver}(\text{Off}) = (\text{dcmd}_{out} = \text{no-cmd})$$
$$\mathbb{M}_{Driver}(\text{Resettable}) = (\text{dcmd}_{out} = \text{no-cmd})$$
$$\mathbb{M}_{Driver}(\text{TurningOn}) = \text{True}$$
$$\mathbb{M}_{Driver}(\text{TurningOff}) = \text{True}$$
$$\mathbb{M}_{Driver}(\text{ResettingOn}) = \text{True}$$
$$\mathbb{M}_{Driver}(\text{ResettingOff}) = \text{True}$$
$$\mathbb{M}_{Driver}(\text{Failed}) = \text{True}.$$

In this example, the transitional modes (*TurningOn*, *TurningOff*, *ResettingOn*, and *ResettingOff*) are assumed to be unconstrained, i.e. the constraint for each of these modes is **True**.

The transitions and associated probabilities for the driver component are specified in Table 6.1, where the probabilities in this table are based on the following:

- there is a 0.1% likelihood of transitioning into *Failed* mode from any other mode, at any time;
- there is a 1% likelihood of transitioning into the *Resettable* fault mode from any nominal mode, at any time;
- the following uniform probability density functions are associated with each timed transition:

$$\mathbf{p}_{\tau_6}(t) = 10 \text{ for } 0 \leq t \leq 0.1, = 0 \text{ otherwise},$$
$$\mathbf{p}_{\tau_{14}}(t) = 10 \text{ for } 0 \leq t \leq 0.1, = 0 \text{ otherwise},$$

$$\mathbf{p}_{\tau_{22}}(t) = 5 \text{ for } 0 \leq t \leq 0.2, = 0 \text{ otherwise},$$
$$\mathbf{p}_{\tau_{26}}(t) = 10 \text{ for } 0 \leq t \leq 0.1, = 0 \text{ otherwise};$$

- the likelihood of remaining in a given mode (i.e. taking a self-transition) is computed as one minus the sum of the probabilities of the outgoing fault and timed transitions.

Mode constraints, transitions and probabilities can be similarly expressed for the valve component.

The following modal rewards are assigned:

$$\mathbb{R}_{Driver}(\text{Off}) = 0,$$
$$\mathbb{R}_{Driver}(\text{On}) = 0,$$
$$\mathbb{R}_{Driver}(\text{TurningOn}) = 0,$$
$$\mathbb{R}_{Driver}(\text{TurningOff}) = 0,$$
$$\mathbb{R}_{Driver}(\text{ResettingOn}) = 0,$$
$$\mathbb{R}_{Driver}(\text{ResettingOff}) = 0,$$
$$\mathbb{R}_{Driver}(\text{Resettable}) = -10,$$
$$\mathbb{R}_{Driver}(\text{Failed}) = -20,$$
$$\mathbb{R}_{Valve}(\text{Closed}) = 0,$$
$$\mathbb{R}_{Valve}(\text{Open}) = -5,$$
$$\mathbb{R}_{Valve}(\text{Stuck-open}) = -20,$$
$$\mathbb{R}_{Valve}(\text{Stuck-closed}) = -10.$$

Note that repairable fault modes, such as *(Driver = Resettable)*, are given a lower reward than nominal modes, to bias mode reconfiguration toward repairing them whenever possible. For the valve, the *Closed* mode is given a higher reward than the *Open* mode, to bias the system toward closing any valves that need not be open in order to achieve a given goal (thus minimizing loss of propellant, for instance).

Table 6.1: Transitions and transition probabilities for the driver component.

	Transition τ_{Driver}	Probability $\mathbf{P}_{\tau_{Driver}}(t_D)$
τ_1	$\langle$Off, (dcmd$_{in}$ = on), TurningOn$\rangle$	0.989
τ_2	$\langle$Off, (dcmd$_{in}$ = on), Off$\rangle$	0.989
τ_3	$\langle$Off, True, Resettable$\rangle$	0.01
τ_4	$\langle$Off, True, Failed$\rangle$	0.001
τ_5	$\langle$TurningOn, True, TurningOn$\rangle$	$0.989(1 - \int_0^{t_D} \mathbf{p}_{\tau_6}(t)dt)$
τ_6	$\langle$TurningOn, True, On$\rangle$	$0.989 \int_0^{t_D} \mathbf{p}_{\tau_6}(t)dt$
τ_7	$\langle$TurningOn, True, Resettable$\rangle$	0.01
τ_8	$\langle$TurningOn, True, Failed$\rangle$	0.001
τ_9	$\langle$On, (dcmd$_{in}$ = off), TurningOff$\rangle$	0.989
τ_{10}	$\langle$On, (dcmd$_{in}$ = off), On$\rangle$	0.989
τ_{11}	$\langle$On, True, Resettable$\rangle$	0.01
τ_{12}	$\langle$On, True, Failed$\rangle$	0.001
τ_{13}	$\langle$TurningOff, True, TurningOff$\rangle$	$0.989(1 - \int_0^{t_D} \mathbf{p}_{\tau_{14}}(t)dt)$
τ_{14}	$\langle$TurningOff, True, Off$\rangle$	$0.989 \int_0^{t_D} \mathbf{p}_{\tau_{14}}(t)dt$
τ_{15}	$\langle$TurningOff, True, Resettable$\rangle$	0.01
τ_{16}	$\langle$TurningOff, True, Failed$\rangle$	0.001
τ_{17}	$\langle$Resettable, (dcmd$_{in}$ = off) $\vee$ (dcmd$_{in}$ = reset), Resettable$\rangle$	0.999
τ_{18}	$\langle$Resettable, (dcmd$_{in}$ = reset), ResettingOn$\rangle$	0.999
τ_{19}	$\langle$Resettable, (dcmd$_{in}$ = off), ResettingOff$\rangle$	0.999
τ_{20}	$\langle$Resettable, True, Failed$\rangle$	0.001
τ_{21}	$\langle$ResettingOn, True, ResettingOn$\rangle$	$0.989(1 - \int_0^{t_D} \mathbf{p}_{\tau_{22}}(t)dt)$
τ_{22}	$\langle$ResettingOn, True, On$\rangle$	$0.989 \int_0^{t_D} \mathbf{p}_{\tau_{22}}(t)dt$
τ_{23}	$\langle$ResettingOn, True, Resettable$\rangle$	0.01
τ_{24}	$\langle$ResettingOn, True, Failed$\rangle$	0.001
τ_{25}	$\langle$ResettingOff, True, ResettingOff$\rangle$	$0.989(1 - \int_0^{t_D} \mathbf{p}_{\tau_{26}}(t)dt)$
τ_{26}	$\langle$ResettingOff, True, Off$\rangle$	$0.989 \int_0^{t_D} \mathbf{p}_{\tau_{26}}(t)dt$
τ_{27}	$\langle$ResettingOff, True, Resettable$\rangle$	0.01
τ_{28}	$\langle$ResettingOff, True, Failed$\rangle$	0.001
τ_{29}	$\langle$Failed, True, Failed$\rangle$	1

To complete the specification of the Timed Constraint Automata for the two components, the following initial mode probabilities are specified:

$$\mathbf{P}_{\Theta_{Driver}}(\text{Off}) = .9,$$
$$\mathbf{P}_{\Theta_{Driver}}(\text{On}) = .1,$$
$$\mathbf{P}_{\Theta_{Valve}}(\text{Closed}) = .9,$$
$$\mathbf{P}_{\Theta_{Valve}}(\text{Open}) = .1,$$

with all other modes having zero initial probability.

For the TCCA composed of the driver and valve Timed Constraint Automata, the set of control variables is $\Pi^c = \{\text{dcmd}_{in}\}$, the set of observable variables is $\Pi^o = \{\text{flow}_{in}, \text{flow}_{out}\}$, and the set of dependent variables is $\Pi^d = \{\text{dcmd}_{out}, \text{vcmd}_{in}\}$.

Finally, the component interconnections are given by:

$$\mathbb{Q} = (\text{dcmd}_{out} = \text{vcmd}_{in}).$$

6.1.4 Feasible Trajectories of a TCCA

Next, consider the set of plant trajectories that are *feasible*, that is, trajectories in which each pair of sequential states is consistent with a TCCA. Discussion of the computation of the most likely plant state, given a set of observations, is deferred to the next section. Recall that, in the interleaving model of computation, each execution cycle consists of an instantaneous "discrete" event and a "continuous" phase in which time advances by some amount δ. δ, the length of an execution cycle, need not remain constant from one cycle to the next. As mentioned in Section 5.3, δ is determined by the amount of time required for the Timed Model-based Executive to complete its control sequencer and deductive controller operations. From the point of view of the deductive controller, the discrete events correspond to transitions between states in $\mathcal{P}$.

As discussed in Section 4.1, the assumption is made that execution cycles are sufficiently short that the state does not change more than once per cycle (i.e., each component only takes one transition per cycle). This assumption is built into the definition of a feasible trajectory, which takes a single step along an enabled transition at each execution cycle. This assumption is necessary to ensure correct operation of the deductive controller: in particular, the current implementation of mode estimation only searches over the states that are reachable via a single transition. Relaxing this assumption would significantly increase the size of the state space that mode estimation would need to search.

In the following definition of a feasible trajectory, a clock initialization vector $t^{init}(i)$ is an n-tuple defining an absolute clock initialization time $t_j^{init}(i)$ for each clock variable, and $t^{abs}(i)$ is specified as the latest reading of the absolute system clock.

Given sequences of control variable assignments $[\mu^{(0)}, \mu^{(1)}, \ldots]$ and absolute system times $[t^{abs}(0), t^{abs}(1), \ldots]$, a *feasible trajectory* of a plant $\mathcal{P}$ is defined by sequences of plant states $[s^{(0)}, s^{(1)}, \ldots]$ and clock initialization times $[t^{init}(0), t^{init}(1), \ldots]$, such that:

1. $s^{(0)}$ is a valid initial plant state, i.e. $\mathbf{P}_{\Theta_j}(x_j = v) > 0$ for all assignments $(x_j = v) \in s^{(0)}$;

2. each entry in $t^{init}(0)$ is set to the absolute system clock time at startup, i.e. $t_j^{init}(0) = t^{abs}(0), \forall j$;

3. for each cycle $i \geq 0$, $\langle s^{(i+1)}, t^{init}(i+1) \rangle = Step_{TCCA}(\mathcal{P},\ s^{(i)},\ \mu^{(i)},\ t^{init}(i),\ t^{abs}(i))$.

$Step_{TCCA} : \Sigma(\Pi^m) \times \Sigma(\Pi^c) \times \Sigma(\Pi^t) \rightarrow \Sigma(\Pi^m) \times \Sigma(\Pi^t)$ transitions the TCCA of a plant $\mathcal{P}$, by executing a transition for each of its component automata. In Figure 6-6, an algorithm for $Step_{TCCA}$ is presented, which handles the non-determinism in the model by non-deterministically choosing one of its enabled transitions to take. As mentioned previously, a transition is enabled if its guard and time constraint are both satisfied. For example, consider the driver component in Figure 6-4. Assuming the current mode is *TurningOn* and the current value of clock t_D is 0.05 sec, transitions

$Step_{TCCA}(\mathcal{P}, s^{(i)}, \mu^{(i)}, t^{init}(i), t^{abs}(i)) \rightarrow \langle s^{(i+1)}, t^{init}(i+1)\rangle$ ::

1. **Define new clock initialization vector.** For each clock variable, set $newInit_j = t_j^{init}(i)$.

2. **Update clocks.** For each clock variable, set $t_j(i) = t^{abs} - t_j^{init}(i)$.

3. **Compute constraint store.** Take the constraint store to be the conjunction of the interconnection constraints and all state constraints:

$$C_M := \mathbb{Q} \wedge \bigwedge_{(x_j = v) \in s^{(i)}} \mathbb{M}_j\,(x_j = v)$$

4. **Identify enabled transitions.** For each component mode variable assignment $x_j = v$, identify as enabled any transition $\tau = \langle v, g, v'\rangle \in \mathbb{T}_j$, for which g is satisfied by the constraint store C_M and the current control variable assignments $\mu^{(i)}$, and for which $\mathbf{P}_\tau > 0$, given the current value of $t_j(i)$ (for transitions with time constraints, this implies that $t_j(i)$ satisfies the time constraint).

5. **Take transitions.** Only one transition can be taken for each component mode variable; when more than one transition is enabled, non-deterministically choose one of them to be taken. For each transition taken, set $newMode_j = v'$.

6. **Assert clock initializations.** For each component mode variable, compare the current and next assignments: if $v' \neq v$, assert clock initialization ($t_j := 0$) by setting $newInit_j = t^{abs}$.

7. **Return feasible next state and clock initialization.**

$$\begin{aligned} s^{(i+1)} &= \bigcup_j \{(x_j = newMode_j)\}, \\ t^{init}(i+1) &= \bigcup_j \{newInit_j\}. \end{aligned}$$

Figure 6-6: $Step_{TCCA}$algorithm.

τ_5, τ_6, τ_7 and τ_8 would all be enabled (see Table 6.1). However, if the current mode is *TurningOn* and the current value of clock t_D is 0.1 sec, the probability of self-transition τ_5 drops to zero; at this point, the only enabled transitions are the outgoing timed transition τ_6, and the two fault transitions τ_7 and τ_8. Note that $Step_{TCCA}$ does not explicitly track the clock interpretations (values of each clock variable). Rather, it tracks the initialization time of each clock (in absolute time); the current value of clock t_j is derived in each cycle by taking the difference between the current absolute time $t^{abs}(i)$, and the initialization time $t_j^{init}(i)$.

The above definition of a feasible TCCA trajectory is based on the definition of feasible trajectories of CCA: $Step_{TCCA}$ extends the untimed $Step_{CCA}$ algorithm presented in [90], by adding steps 1, 2, and 6, and modifying the notion of enabled transitions to include transitions whose time constraints are satisfied by the current

clock values. As for CCA, ${Step_{TCCA}}^N$ denotes a variant of the step function that takes only nominal transitions. A trajectory that involves only nominal transitions is called a *nominal trajectory.*

6.2 Mode Estimation

In this section, the ME capability of the deductive controller is described. Recall that the role of ME is to estimate the current set of component modes that comprise the state of a physical plant. The ME algorithm used by the Timed Model-based Executive builds off the $Step_{TCCA}$ algorithm presented in the previous section, using the transition probabilities and the observations from the system's execution to resolve the non-determinism in the plant model. This section begins by defining the notion of *consistency* with respect to the constraints imposed by an observation sequence. It then describes belief state update for plants modeled as TCCA. Finally, the ME algorithm is presented, which provides a tractable approximation of belief state update.

6.2.1 Consistent Executions of a TCCA

In this section, the notion of feasible trajectories presented in Section 6.1.4 is extended to define state trajectories that are consistent with respect to a sequence of plant observations. Given sequences of control variable assignments $[\mu^{(0)}, \mu^{(1)}, \ldots]$, observable variable assignments $[o^{(1)}, o^{(2)}, \ldots]$, and absolute system times $[t^{abs}(0), t^{abs}(1), \ldots]$, a *consistent trajectory* of a plant $\mathcal{P}$ is defined by sequences of plant states $[s^{(0)}, s^{(1)}, \ldots]$ and clock initialization times $[t^{init}(0), t^{init}(1), \ldots]$, such that:

1. $s^{(0)}$ is a valid initial plant state, i.e. $\mathbf{P}_{\Theta_j}(x_j = v) > 0$ for all assignments $(x_j = v) \in s^{(0)}$;

2. each entry in $t^{init}(0)$ is set to the absolute system clock time at startup, i.e. $t^{init}_j(0) = t^{abs}(0), \forall j$;

3. for each cycle $i \geq 0$, $\langle s^{(i+1)}, t^{init}(i+1)\rangle = ConsistentState_{TCCA}(\mathcal{P},\ s^{(i)},\ \mu^{(i)},\ o^{(i+1)},\ t^{init}(i),\ t^{abs}(i))$.

$ConsistentState_{TCCA} : \Sigma(\Pi^m) \times \Sigma(\Pi^c) \times \Sigma(\Pi^o) \times \Sigma(\Pi^t) \rightarrow \Sigma(\Pi^m) \times \Sigma(\Pi^t)$ transitions the TCCA of a plant $\mathcal{P}$, by executing a transition for each of its component automata, and by comparing the next state with the observations. Figure 6-7 presents an algorithm for $ConsistentState_{TCCA}$, based on the $Step_{TCCA}$ algorithm from Section 6.1.4. $ConsistentState_{TCCA}$ adds two steps to $Step_{TCCA}$. In step 6, it computes the constraint store corresponding to the modes in the proposed next state, $s^{(i+1)}$. In step 7, it makes sure that the observations $o^{(i+1)}$ are consistent with this constraint store and the proposed next state $s^{(i+1)}$.

Now that consistent trajectories have been defined, the probabilistic nature of timed plant models can be addressed. The following section describes how the TCCA's transition probabilities are used to estimate the state of the plant.

6.2.2 Belief State Update for TCCA

Recall from Section 4.3.2 that ME for a factored POSMDP can be framed as a variant of belief state update. In this section, the general belief state update approach is formalized for timed plant models expressed as TCCA.

At this point, the probabilistic nature of the transitions in a TCCA is considered. As described in Section 6.1.2, probabilities $\mathbf{P}_\tau(t_j)$ have been specified, corresponding to the likelihood that a transition τ will be taken for any given value of the component's clock variable t_j. This probability information can be used to compute, in each execution cycle, the belief state associated with each possible plant state and clock interpretation.

The key insight is that a factored POSMDP, as encoded in a TCCA, can be mapped to a POMDP by augmenting the plant state with the clock interpretation. Thus, the *system state* $\mathbf{s}^{(i)}$ is specified by the current assignments to each component's state variable, as well as the current clock interpretation:

$$\mathbf{s}^{(i)} = \langle s^{(i)}, t(i)\rangle.$$

$ConsistentState_{TCCA}(\mathcal{P}, s^{(i)}, \mu^{(i)}, o^{(i+1)}, t^{init}(i), t^{abs}(i)) \rightarrow \langle s^{(i+1)}, t^{init}(i+1)\rangle$ ::

1. **Define new clock initialization vector.** For each clock variable, set $newInit_j = t_j^{init}(i)$.

2. **Update clocks.** For each clock variable, set $t_j(i) = t^{abs} - t_j^{init}(i)$.

3. **Compute constraint store.** Take the constraint store to be the conjunction of the interconnection constraints and all state constraints:
$$C_M^{(i)} := \mathbb{Q} \wedge \bigwedge_{(x_j = v) \in s^{(i)}} \mathbb{M}_j(x_j = v)$$

4. **Identify enabled transitions.** For each component mode variable assignment $x_j = v$, identify as enabled any transition $\tau = \langle v, g, v'\rangle \in \mathbb{T}_j$, for which g is satisfied by the constraint store $C_M^{(i)}$ and the current control variable assignments $\mu^{(i)}$, and for which $\mathbf{P}_\tau > 0$, given the current value of $t_j(i)$ (for transitions with time constraints, this implies that $t_j(i)$ satisfies the time constraint).

5. **Choose a transition for each component.** For each component, non-deterministically choose an enabled transition to be taken. For each transition taken, set $newMode_j = v'$.

6. **Compute constraint store for proposed next state.** Let $s^{(i+1)} = \bigcup_j \{(x_j = newMode_j)\}$. The constraint store for the next state is the conjunction of the interconnection constraints and all state constraints for the new modes:
$$C_M^{(i+1)} := \mathbb{Q} \wedge \bigwedge_{(x_j = v') \in s^{(i+1)}} \mathbb{M}_j(x_j = v')$$

7. **Check consistency of proposed next state.** Unless $s^{(i+1)} \wedge C_M^{(i+1)} \wedge o^{(i+1)}$ is consistent, return to step 5 and select a different set of enabled transitions to take.

8. **Assert clock initializations.** For each component mode variable, compare the current and next assignments: if $v' \neq v$, assert clock initialization ($t_j := 0$) by setting $newInit_j = t^{abs}$.

9. **Return feasible next state and clock initialization.**
$$\begin{aligned} s^{(i+1)} &= \bigcup_j \{(x_j = newMode_j)\}, \\ t^{init}(i+1) &= \bigcup_j \{newInit_j\}. \end{aligned}$$

Figure 6-7: $ConsistentState_{TCCA}$ algorithm.

Though multiple trajectories in the system state space can lead to the same physical plant state, these trajectories do not necessarily assign the same value to each clock variable. Since state transitions are conditioned on clock constraints as well as control actions, this means that system states that assign the same values to plant state variables but different values to clock variables can lead to divergent state evolutions. It would therefore be inappropriate to merge these multiple trajectories during belief update, unless the full system state is the same (that is, both the state and clock variables have the same values). Consequently, even though the likelihood of a

plant state is computed by summing the probabilities of trajectories leading to that state, these trajectories are tracked separately unless they correspond to identical system states, that is, unless they also have common clock variable assignments.

Once the timed problem has been mapped to an untimed (i.e. Markovian) problem, the belief state at cycle $i+1$ can be computed from the belief state and control actions at cycle i and observations at $i+1$ using the standard belief update equations:

$$\begin{aligned} p^{(\bullet i+1)}[\mathbf{s}_l] &= \sum_{k=1}^{n} p^{(i\bullet)}[\mathbf{s}_k]\mathbf{P}_{\mathbb{T}}(\mathbf{s}_l \mid \mathbf{s}_k, \mu^{(i)}) \\ p^{(i+1\bullet)}[\mathbf{s}_l] &= p^{(\bullet i+1)}[\mathbf{s}_l]\frac{\mathbf{P}_{\mathbb{O}}(o^{(i+1)} \mid \mathbf{s}_l)}{\sum_{k=1}^{n} p^{(\bullet i+1)}[\mathbf{s}_k]\mathbf{P}_{\mathbb{O}}(o^{(i+1)} \mid \mathbf{s}_k)}, \end{aligned}$$

where the prior probability $p^{(\bullet i+1)}[\mathbf{s}_l]$ is conditioned on all observations up to $o^{(i)}$, and the posterior probability (belief state) $p^{(i+1\bullet)}[\mathbf{s}_l]$ is also conditioned on the latest observation $o^{(i+1)}$. $\mathbf{P}_{\mathbb{T}}(\mathbf{s}_l \mid \mathbf{s}_k, \mu^{(i)})$ is defined as the probability that $\mathcal{P}$ transitions from system state $\mathbf{s}_k$ to system state $\mathbf{s}_l$, given control actions $\mu^{(i)}$. $\mathbf{P}_{\mathbb{O}}(o^{(i+1)} \mid \mathbf{s}_l)$ is the probability that observation $o^{(i+1)}$ is received in system state $\mathbf{s}_l$. The initial belief state is computed based on $p^{(\bullet 0)}[\mathbf{s}_l] = \mathbf{P}_{\Theta}(\mathbf{s}_l)$, where $\mathbf{P}_{\Theta}(\mathbf{s}_l) = \mathbf{P}_{\Theta}(s_l)$ if clock interpretation t_l assigns zero to each plant clock, and $\mathbf{P}_{\Theta}(\mathbf{s}_l) = 0$ otherwise. To complete the presentation of belief update for TCCA, the transition probability $\mathbf{P}_{\mathbb{T}}$ and observation probability $\mathbf{P}_{\mathbb{O}}$ must be defined.

Transition Probability

To calculate the transition probability $\mathbf{P}_{\mathbb{T}}(\mathbf{s}_l \mid \mathbf{s}_k, \mu^{(i)})$ of a plant TCCA, recall that a plant transition $\mathbb{T}$ is composed of a set of component transitions, one for each component mode $(x_j = v)$. The key assumption is made that component transition probabilities are conditionally independent, given the current plant state. This is analogous to the failure independence assumptions made by GDE [19], Sherlock [20] and Livingstone [83], and is a reasonable assumption for most engineered systems. Hence, the plant transition probability is computed as a product of the component

transition probabilities:

$$\mathbf{P}_{\mathbb{T}}(\mathbf{s}_l \mid \mathbf{s}_k, \mu^{(i)}) = \prod_{x_j \in \Pi^m} \mathbf{P}_{\mathbb{T}_j}(x_j^{(i+1)} = v' \mid x_j^{(i)} = v, \mu^{(i)}, t_j^{(i)}),$$

where $\mathbf{P}_{\mathbb{T}_j}(x_j^{(i+1)} = v' \mid x_j^{(i)} = v, \mu^{(i)}, t_j^{(i)})$ is the probability $\mathbf{P}_{\tau_j}(t_j^{(i)})$ for the transition from mode $(x_j = v) \in s_k$ in cycle i to mode $(x_j = v') \in s_l$ in cycle $i+1$ enabled by the current constraint store and the control action $\mu^{(i)}$. [1]

As mentioned previously, a TCCA specifies in $\mathbf{P}_{\mathbb{T}_j}$ a probability for each transition τ_j, given the current clock value $t_j^{(i)}$. For an enabled timed transition τ_j from mode v to mode v', the component transition probability $\mathbf{P}_{\tau_j}(t_j^{(i)})$ is obtained by integrating from $t_j^{(i-1)}$ to $t_j^{(i)}$ the conditional probability density function $\mathbf{p}_{\tau_j | x_j^{(i-1)}=v}$, given that the transition had not yet occurred at $t_j^{(i-1)}$. This conditional probability density essentially rescales the original probability density function $\mathbf{p}_{\tau_j}$ over the time interval $[t_j^{(i)},\infty)$. In other words, for an enabled timed transition:

$$\mathbf{P}_{\mathbb{T}_j}(x_j = v' \mid x_j = v, \mu^{(i)}, t_j^{(i)}) = \mathbf{P}_{\tau_j}(t_j^{(i)}) = \int_{t_j^{(i-1)}}^{t_j^{(i)}} \mathbf{p}_{\tau_j | x_j^{(i-1)}=v} dt_j.$$

The trellis diagrams in Figures 6-8 and 6-9 depict the possible state evolutions for the simple engine model from Figure 6-5, assuming two different sequences of control actions $\mu^{(i)}$. In Figure 6-8, the engine starts in *Off* mode, is issued the command *cmd = standby* in the first cycle, and is given no further commands. Note the changes in the probabilities associated with the transitions, as time advances through the interval that satisfies the time constraint on the transition from *Heating* mode to *Standby* mode. At time $t_1 + 60$, the only enabled outgoing transitions from *Heating* mode are the timed transition to *Standby* and the fault transition to *Failed* mode. Because the upper bound has been reached on the time constraint for the *Heating* to *Standby* transition, the self-transition from *Heating* mode back to itself is no longer

[1] This assumes each Timed Constraint Automaton specifies only one transition from any given source mode to any given target mode. If more than one transition from source to target is enabled, $\mathbf{P}_{\mathbb{T}_j}(x_j = v' \mid x_j = v, \mu^{(i)}, t_j^{(i)})$ would be computed by summing the probabilities of these "parallel" transitions.

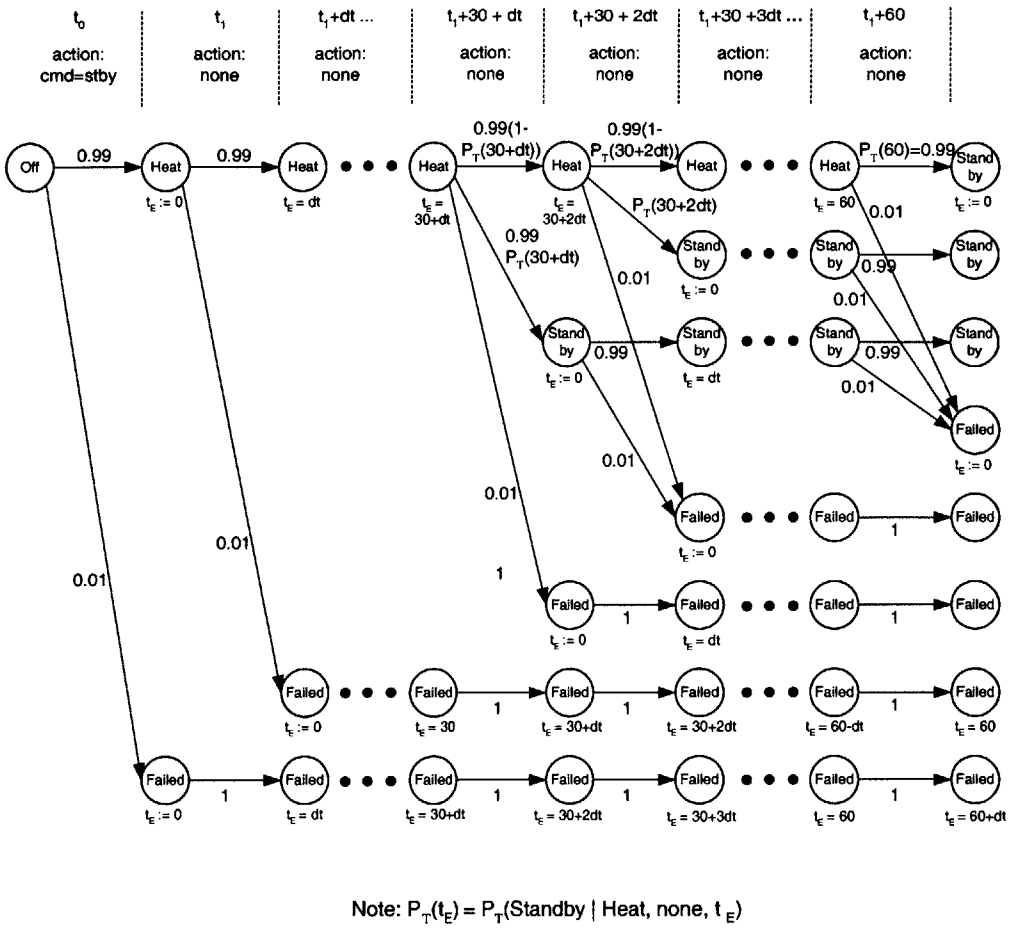

Figure 6-8: Trellis diagram showing the possible state evolutions for the simple spacecraft engine model, for the sequence of control actions $\{cmd = stby,\ none,\ none, \ldots\}$.

enabled: nominal behavior of the engine dictates that the engine must transition to *Standby*. In Figure 6-9, the command $cmd = off$ is issued at time $t_1 + 30 + 3dt$. In this case, transitions back to *Off* mode are taken (in the nominal case), whether or not the timed transition from *Heating* to *Standby* mode has been taken by that time.

Observation Probability

The observation function $\mathbf{P}_{\mathbb{O}}(o^{(i+1)} \mid \mathbf{s}_l)$ is calculated from the TCCA, taking an approach similar to that of GDE [19]. Given the constraint store $C_M^{(i+1)}$ for s_l, computed in Step 6 of $ConsistentState_{TCCA}$, each observation in $o^{(i+1)}$ is tested for entailment

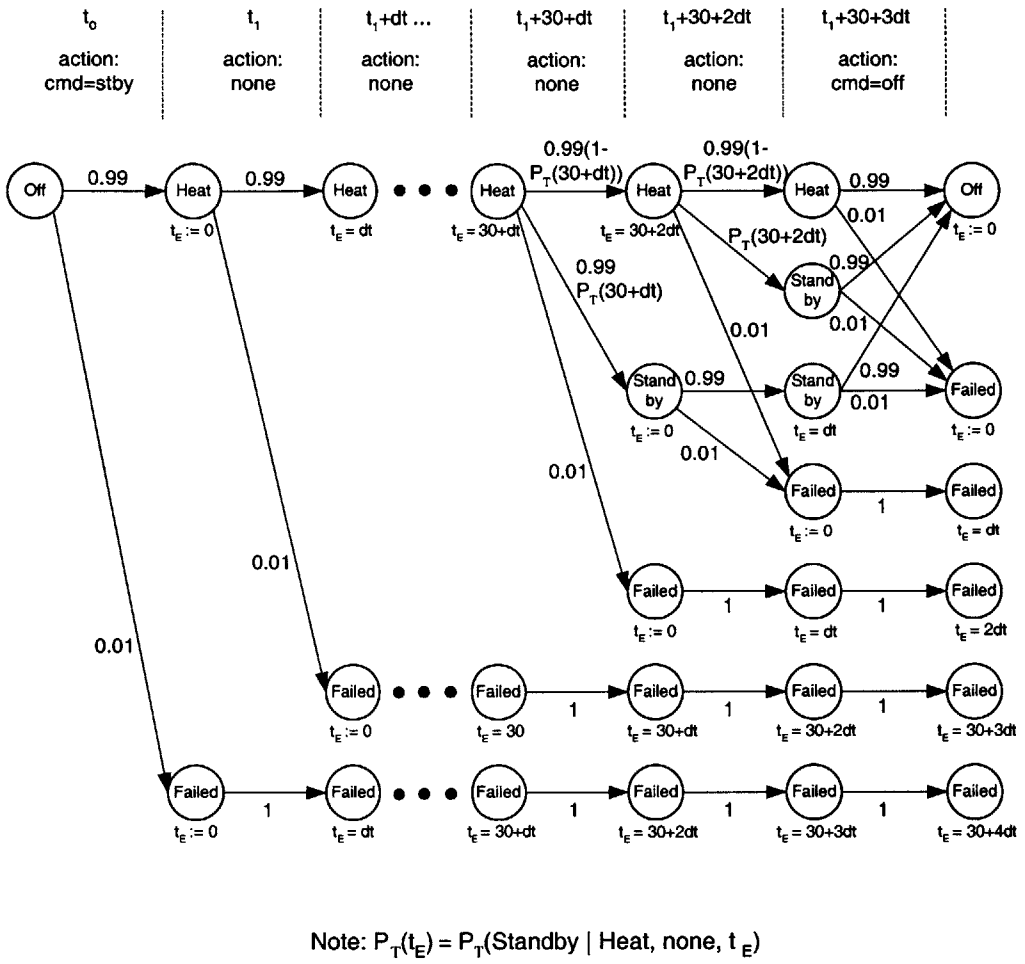

Figure 6-9: Trellis diagram showing the possible state evolutions for the simple spacecraft engine model, for the sequence of control actions $\{cmd = stby,\ none,\ none,\ \ldots,\ cmd = off\}$.

or refutability by the conjunction of $C_M^{(i+1)}$ and s_l, giving $\mathbf{P}_{\mathbb{O}}$ probability 1 or 0, respectively, for that observation. If no prediction is made (i.e. if an observation is neither entailed nor refuted), then an *a priori* distribution on possible values of the observable variable is assumed (e.g., a uniform distribution of $1/n$ for n possible values). This offers a probabilistic bias towards states that predict observations, over states that are merely consistent with the observations. $\mathbf{P}_{\mathbb{O}}(o^{(i+1)} \mid \mathbf{s}_l)$ is computed as the product of the $\mathbf{P}_{\mathbb{O}}$ for all observations in $o^{(i+1)}$. These two definitions for $\mathbf{P}_{\mathbb{T}}$ and $\mathbf{P}_{\mathbb{O}}$ complete the belief update equations for TCCA.

Implementing Belief Update for TCCA

An algorithm for TCCA belief update, $BeliefUpdate_{TCCA}$, is presented in Figure 6-10). $BeliefUpdate_{TCCA}$ computes $p^{(\bullet i+1)}[\mathbf{s}_l]$ for each possible next system state that is consistent with $o^{(i+1)}$. $BeliefUpdate_{TCCA}$ takes in a plant model $\mathcal{P}$, a set of possible current system states $\mathbf{S}^{(i)}$, the posterior probabilities (belief states) $p^{(i\bullet)}$ for each possible current system state, the current system time $t^{abs}(i)$, the control action $\mu^{(i)}$, and the new observation $o^{(i+1)}$. It returns a set of possible next system states $\mathbf{S}^{(i+1)}$ and the posterior probabilities $p^{(i+1\bullet)}$ for every next system state consistent with the latest observations. Rather than keep the current value of each clock variable in the system state, it stores the last initialization time (as measured on the absolute system clock) for each clock. Thus, the current value of each clock variable is computed in each cycle by taking the difference between the initialization time and the current absolute time.

To compute the probability associated with each plant state (independent of the clock values), the probabilities of all returned system states in $\mathbf{S}^{(i+1)}$ that have the same state variable assignments are summed. The resulting most likely plant state is taken as the current state estimate, for the purposes of control program execution.

This algorithm for TCCA-based belief state update provides an "ideal" solution for the mode estimation problem (i.e., an implementation of the semantic model described in Section 4.3.2). Various alternative approaches for performing model-based state estimation on timed systems are documented in the literature. For example, Largouet and Cordier [54] have proposed an approach that performs consistency-based diagnosis on Timed Automata via model-checking. More specifically, their approach uses the Kronos [10] model-checking engine's reachability analysis capability to compute diagnostic trajectories explaining a sequence of observations. Their modeling representation is similar to TCCA, in that it defines a system as a composition of component automata, which capture timed behaviors of the components, such as transition latencies. However, unlike the constraint-based encoding of TCCA, their Timed Automata model is fluent-based, and thus does not support more gen-

$BeliefUpdate_{TCCA}(\mathcal{P}, \mathbf{S}^{(i)}, p^{(i\bullet)}[\mathbf{S}^{(i)}], \mu^{(i)}, o^{(i+1)}, t^{abs}(i)) \rightarrow (\mathbf{S}^{(i+1)}, p^{(i+1\bullet)}[\mathbf{S}^{(i+1)}])$ ::

1. Initialize *targetStates* and *targetProbs* to empty sets. These sets accumulate the target system states, and the probabilities associated with each target system state.

2. For each system state $\mathbf{s}^{(i)} = \langle s^{(i)}, t^{init}(i) \rangle \in \mathbf{S}^{(i)}$:

   - Initialize *newMode*, *newInit* and *newProb* to empty sets. These sets will hold the target modes, clock variable initialization times and transition probabilities, for each enabled transition.
   - **Update clocks.** For each clock variable, set $t_j = t^{abs} - t_j^{init}(i)$.
   - **Compute constraint store.**
   $$C_M^{(i)} = \mathbb{Q} \wedge \bigwedge_{(x_j = v) \in s^{(i)}} \mathbb{M}_j(x_j = v)$$
   - **Identify enabled transitions.** For each component mode variable assignment $(x_j = v) \in s^{(i)}$, identify as enabled any transition $\tau = \langle v, g, v' \rangle \in \mathbb{T}_j$, for which g is satisfied by the constraint store $C_M^{(i)}$ and the current control variable assignments $\mu^{(i)}$, and for which $\mathbf{P}_\tau > 0$, given the current value of t_j (for transitions with time constraints, this implies that t_j satisfies the time constraint). For each enabled transition τ, let $newMode_j(\tau) = v'$ and $newProb_j(\tau) = \mathbf{P}_\tau(t_j)$.
   - **Assert clock initializations.** For each enabled transition τ, if $v' \neq v$, assert clock initialization $(t_j := 0)$ by letting $newInit_j(\tau) = t^{abs}$. If $v' = v$, let $newInit_j(\tau) = t_j^{init}(i)$.
   - **Compute set of target states and probabilities.** Add to ordered set *targetStates* each element $targetStates_k = \langle (x_1 = v_1'), \ldots, (x_n = v_n'), t_1^{init}, \ldots, t_n^{init} \rangle$, where $\langle v_1', \ldots, v_n' \rangle \in \bigotimes_{j=1..n} newMode_j$, and $\langle t_1^{init}, \ldots, t_n^{init} \rangle \in \bigotimes_{j=1..n} newInit_j$. Add to ordered set *targetProbs* each element $targetProbs_k = p^{(i\bullet)}[\mathbf{s}^{(i)}] \prod_{j=1}^{n} p_j$, where $\langle p_1, \ldots, p_n \rangle \in \bigotimes_{j=1..n} newProb_j$.

3. **Consolidate probabilities for common targets.** Let $\mathbf{S}^{(i+1)} = \bigcup_k \{targetStates_k\}$ be the set of unique target system states, and let $p^{(\bullet i+1)}[\mathbf{s}^{(i+1)}]$ be the sum of all probabilities $targetProbs_k$ associated with the same target state $\mathbf{s}^{(i+1)} \in \mathbf{S}^{(i+1)}$.

4. For each $\mathbf{s}^{(i+1)} = \langle s^{(i+1)}, t^{init}(i+1) \rangle \in \mathbf{S}^{(i+1)}$:

   - **Compute constraint store for proposed next state.** The constraint store for the next state is the conjunction of the interconnection constraints and all state constraints for the new modes:
   $$C_M^{(i+1)} := \mathbb{Q} \wedge \bigwedge_{(x_j = v') \in s^{(i+1)}} \mathbb{M}_j(x_j = v').$$
   - **Compute observation probability.**
   $$\mathbf{P_O}(o^{(i+1)} \mid \mathbf{s}^{(i+1)}) = \prod_{l=1}^{n_o} \mathbf{P_O}(o_l^{(i+1)} \mid \mathbf{s}^{(i+1)}), \text{ where:}$$
   $$\mathbf{P_O}(o_l^{(i+1)} \mid \mathbf{s}^{(i+1)}) = 1 \text{ if } s^{(i+1)} \wedge C_M^{(i+1)} \models o_l^{(i+1)},$$
   $$\mathbf{P_O}(o_l^{(i+1)} \mid \mathbf{s}^{(i+1)}) = 0 \text{ if } s^{(i+1)} \wedge C_M^{(i+1)} \models \neg o_l^{(i+1)},$$
   $$\mathbf{P_O}(o_l^{(i+1)} \mid \mathbf{s}^{(i+1)}) = 1/m \text{ otherwise, and}$$
   $$m = \text{ number of possible values for the observation } o_l^{(i+1)}.$$
   - **Condition on latest observation.** Compute the posterior probability associated with $\mathbf{s}^{(i+1)}$ by multiplying by the observation probability:
   $$p^{(i+1\bullet)}[\mathbf{s}^{(i+1)}] = p^{(\bullet i+1)}[\mathbf{s}^{(i+1)}] \mathbf{P_O}(o^{(i+1)} \mid \mathbf{s}^{(i+1)}).$$

   Note that the normalization by the overall observation probability $\mathbf{P_O}(o^{(i+1)})$ is omitted here. The posterior probability is therefore not exact, but still provides an accurate likelihood ordering.

5. **Return next system states and probabilities.** Return $\mathbf{S}^{(i+1)}$ and $p^{(i+1\bullet)}[\mathbf{S}^{(i+1)}]$.

Figure 6-10: $BeliefUpdate_{TCCA}$ algorithm.

eral behavior specifications based on propositional logic constraints. Furthermore, Largouet and Cordier model fault transitions deterministically, by conditioning transitions into fault modes on explicit event labels. Non-determinism is folded into the system through its observation model, where observations can be modeled as *imprecise*, i.e. an observation is described by a set of possible fluents. Unlike the TCCA, their method provides no mechanism for diagnostic discrimination based on likelihood of failure. To address this limitation, Largouet and Cordier have extended their approach [53] through the adoption of Probabilistic Timed Automata. A comparison between their PTA model and TCCA was made in Section 6.1.2.

One benefit of diagnosis methods based on model-checking is that they do not require the assumption that only a single plant transition is taken in each diagnostic cycle. Model checkers like Kronos consider any trajectory (single- or multi-step) through the Timed Automaton model that leads to a state explaining the latest observation. However, this advantage comes at the expense of reactivity. Furthermore, diagnosis methods based on model-checking are exhaustive, in that they compute all consistent nominal and faulty trajectories at each diagnosis step. Consequently, these approaches become computationally expensive for complex processes (i.e. Timed Automata with large state spaces and many different fault behaviors). In order to perform estimation and diagnosis on-line during time-critical sequence execution, it becomes necessary to use approximate methods that incrementally compute the most-likely states of the system. Describing an approximation to the intractable belief state update process for TCCA is the topic of the following section.

6.2.3 Approximate Belief Update for TCCA

As illustrated in Figures 6-8 and 6-9, augmenting the plant state with the current clock interpretations results in a significant expansion of the state space. This expansion is associated with the presence of non-determinism in the model, in the form of timed nominal transitions and fault transitions. For instance, in each execution cycle where a timed transition is enabled, the non-deterministic nature of the timed transition leads to a new branch in the Trellis diagram, even in the absence of a control action.

Similarly, additional branching results from the non-determinism associated with fault transitions, as seen by the expansion to a new *failed* node at each step in the Trellis diagram. Even for a single-component system, as depicted in Figures 6-8 and 6-9, the number of additional branches in the Trellis diagram that appear over a given time horizon $\mathcal{T}$ is on the order of $m \cdot \mathcal{T}/\delta$, in the worst-case, where m is the number of possible states at the start of the time horizon, and δ is the average execution time step. For the full system model, this implies a worst case state expansion on the order of $(m \cdot \mathcal{T}/\delta)^n$ over the time horizon $\mathcal{T}$, where n is the number of components in the system.

Because it computes the full belief state (i.e., the probability associated with each possible system state) at each step, the *BeliefUpdateTCCA* algorithm is very costly, in terms of memory usage and computation time. In particular, step 2 iterates over each system state in the current belief state and step 4 iterates over each system state in the new belief state. For embedded systems with severe memory and computational resource constraints, such as spacecraft, a state estimation approach that limits the amount of online search performed and the number of possible states tracked must be used. To this end, a key assumption is made: *the probabilities of a limited number of most-likely system states are assumed to dominate the probabilities of other system states.* Recent implementations of mode estimation (for untimed plant models) have leveraged this type of assumption to approximate belief state update by tracking a limited set of most-likely states, from one cycle to the next. For instance, the Livingstone [83] and Titan [90] model-based executives perform mode estimation by identifying the κ best extensions to the current most likely trajectories, as solutions to a shortest path problem. This shortest path problem can be framed as an Optimal Constraint Satisfaction Problem (OCSP), which is solved using the OPSAT algorithm, presented in detail in [67, 91].

An OCSP $\langle \mathbf{x}, f, C \rangle$ is a problem of the form "arg max $f(\mathbf{x})$ subject to $C(\mathbf{x})$," where $\mathbf{x}$ is a vector of decision variables, $C(\mathbf{x})$ is a set of propositional state constraints, and $f(\mathbf{x})$ is a multi-attribute utility function that is *mutually preferentially independent (MPI).* $f(\mathbf{x})$ is MPI if the value of each $x_i \in \mathbf{x}$ that maximizes f is

independent of the values assigned to the remaining variables. Solving an OCSP consists of generating a prefix of the sequence of feasible solutions, ordered by decreasing value of f. A feasible solution assigns to each variable in $\mathbf{x}$ a value from its domain such that $C(\mathbf{x})$ is satisfied. To solve an OCSP, OPSAT tests a leading candidate for consistency against $C(\mathbf{x})$. If it proves inconsistent, OPSAT summarizes the inconsistency (called a *conflict*) and uses the summary to jump over other leading candidates that are similarly inconsistent. OPSAT generates solutions to the OCSP in best-first order, and can thus be used to generate the κ leading solutions.

The Timed Model-based Executive's deductive controller adopts a similar approach to ME. Starting from a current approximate belief state (consisting of the κ most likely system states), it computes at each execution cycle a new approximate belief state, based on the latest control actions, the current system time, and the new observation from the plant. One limitation of this approach is that a low-probability system state may be pruned, which could become very likely after additional evidence is collected. It should be noted, however, that in the current paradigm of flight software design, typical onboard fault protection is designed to handle single faults, or, at worst, dual faults. More complex fault scenarios are generally considered so unlikely as to be unnecessary to model and test for. Thus, by judiciously setting the "number of solutions" parameter κ, the approximate belief state approach provides no less fault coverage than current spacecraft fault protection systems.

Another limitation is that this approach does not add posterior probabilities for multiple separate trajectories leading to the same target system state; instead, the most likely trajectory among these multiple trajectories is taken as the shortest path to the target, and its associated probability is used in the computation of the target system state's probability. Consequently, the resulting target system state "probability" is an underestimate (lower bound) of the true probability of being in that target system state.

Formally, the system state probabilities at cycle $i + 1$ are computed from the system state probabilities and control actions at i and the observations at $i+1$, using

the following shortest-path update equations:

$$
\begin{aligned}
p^{(\bullet i+1)}[\mathbf{s}_l] &= \max_{k=1..\kappa} p^{(i\bullet)}[\mathbf{s}_k]\mathbf{P}_{\mathbb{T}}(\mathbf{s}_l \mid \mathbf{s}_k, \mu^{(i)}) \\
p^{(i+1\bullet)}[\mathbf{s}_l] &= p^{(\bullet i+1)}[\mathbf{s}_l]\frac{\mathbf{P}_{\mathbb{O}}(o^{(i+1)} \mid \mathbf{s}_l)}{\sum_{k=1}^{\kappa} p^{(\bullet i+1)}[\mathbf{s}_k]\mathbf{P}_{\mathbb{O}}(o^{(i+1)} \mid \mathbf{s}_k)}
\end{aligned}
$$

with $\mathbf{P}_{\mathbb{T}}(\mathbf{s}_l \mid \mathbf{s}_k, \mu^{(i)})$ and $\mathbf{P}_{\mathbb{O}}(o^{(i+1)} \mid \mathbf{s}_l)$ defined as above for belief state update.

The *TimedME* algorithm, presented in Figure 6-11, computes this shortest-path approximate solution to the belief state update problem (see Figure 6-11). *TimedME* takes in a plant model $\mathcal{P}$, a set of κ current system states $\mathbf{S}^{(i)}$, the approximate belief state probabilities $p^{(i\bullet)}$ for system states in $\mathbf{S}^{(i)}$, the current system time $t^{abs}(i)$, the control action $\mu^{(i)}$, and the new observation $o^{(i+1)}$. It returns the set of κ most likely next system states $\mathbf{S}^{(i+1)}$ and the approximate belief update probabilities $p^{(i+1\bullet)}$ for each system state in $\mathbf{S}^{(i+1)}$.

Step 1 of *TimedME* extracts from $\mathbf{S}^{(i)}$ each current possible system state $\mathbf{s}^k$, and defines each system state in terms of its plant state s^k and vector t^{init^k} of clock initialization times. Step 2 updates the set of plant clocks, by advancing each clock by the amount of time elapsed since the previous call to *TimedME*. Step 3 frames the enumeration of likely trajectory extensions as an OCSP, as follows.

Recall that each plant transition consists of a single transition for each of its component automata. A solution to the OCSP corresponds to a plant transition τ. Hence, a decision variable τ_j is introduced into τ for each component j. The domain of each decision variable τ_j is the set of possible transitions for component j. $C(\tau)$ is used to encode the constraint that:

- the source state of solution τ and its corresponding constraint store $C_M^{(i)}$ must be consistent with one of the current system states in $\mathbf{S}^{(i)}$ and the control action $\mu^{(i)}$; and
- the target state of τ and its corresponding constraint store $C_M^{(i+1)}$ must be consistent with the observed values $o^{(i+1)}$.

TimedME $(P, \mathbf{S}^{(i)}, p^{(i\bullet)}, \mu^{(i)}, o^{(i+1)}, t^{abs}) \rightarrow (\mathbf{S}^{(i+1)}, p^{(i+1\bullet)})$ ::

1. Let $\mathbf{s}^k = \langle s^k, t^{init^k} \rangle$ be the k-th element in $\mathbf{S}^{(i)} = \langle \mathbf{s}^1, \mathbf{s}^2, \ldots, \mathbf{s}^\kappa \rangle$. s^k is a vector of component mode assignments $(x_j^{(i)} = v)$, and t^{init^k} is a vector of initialization times $t_j^{init^k}$ for each plant clock.

2. **Update plant clocks to reflect elapsed time since last execution cycle:**

   For $k = 1$ to κ, define $t^k = \langle (t_1^k = t^{abs} - t_1^{init^k}), (t_2^k = t^{abs} - t_2^{init^k}), \ldots, (t_n^k = t^{abs} - t_n^{init^k}) \rangle$.

3. **Setup the OCSP** $\langle \tau, f, C \rangle$**:**

   - The vector τ includes a decision variable τ_j for each component of the plant. The domain of τ_j is the set of transitions $\langle v, g, v' \rangle \in \mathbb{T}_j$, where each transition is expressed as a propositional logic formula $\left[(x_j^{(i)} = v) \wedge g \Rightarrow (x_j^{(i+1)} = v')\right]$. Source$(\tau_j)$ and Target(τ_j) denote the source and target modes for component transition τ_j. Source(τ) and Target(τ) denote the source and target plant states associated with plant transition τ.
   - The objective function $f(\tau)$ is the probability of a trajectory into the next system state, that is,

$$\begin{aligned} f(\tau) &= p^{(i\bullet)}[\mathbf{s}^{(i)}] \cdot \mathbf{P}_\tau(\mathbf{s}^{(i+1)} \mid \mathbf{s}^{(i)}, \mu^{(i)}) \\ &= p^{(i\bullet)}[\mathbf{s}^{(i)}] \cdot \prod_{j=1..n} \mathbf{P}_{\tau_j}(v' \mid v, \mu^{(i)}, t_j), \end{aligned}$$

   - $C(\tau)$ encodes the constraint that $\tau \wedge (\bigvee_{k=1..\kappa}(s^k \wedge t^k)) \wedge C_M^{(i)} \wedge \mu^{(i)} \wedge C_M^{(i+1)} \wedge o^{(i+1)}$ must be consistent, where $C_M^{(i)} = \mathbb{Q} \wedge \bigwedge_{j=1..n} \left[\bigwedge_{v \in \mathbb{D}[x_j]} \left[(x_j^{(i)} = v) \Rightarrow \mathbb{M}_j(x_j^{(i)} = v)\right]\right]$.
   - The OPSAT solver requires that all variables have finite domain. Thus, the domain of each time variable t_j is taken to be the discrete and finite set of possible times $\{t_j^1, t_j^2, \ldots, t_j^\kappa\}$.

4. **Compute the** κ **most likely solutions to the OCSP using OPSAT:**

   OPSAT returns solution $\langle \tau^1, \tau^2, \ldots \tau^\kappa \rangle$ and $\langle \mathbf{s'}^1, \mathbf{s'}^2, \ldots, \mathbf{s'}^\kappa \rangle$, where $\mathbf{s'}^k = \langle \text{Target}(\tau^k), t^{init^k} \rangle$, and t^{init^k} is the vector of clock initialization times in the system state $\mathbf{s}^k$ that led to the solution τ^k.

5. **Reset the appropriate clocks in the target system state of each solution:**

   For each solution $\langle \tau^k, \mathbf{s'}^k \rangle$, update the clock initialization times in $\mathbf{s'}^k$ as follows:

   For each component mode variable x_j, compare the assignments $(x_j = v) \in \text{Source}(\tau_j^k)$ and $(x_j = v') \in \text{Target}(\tau_j^k)$:

   if $v' \neq v$, let $t^{init^k}_j(i+1) = t^{abs}$;

   if $v' = v$, let $t^{init^k}_j(i+1)$ remain unchanged.

6. **Return the set of target system states, and the associated probabilities:**

$$\mathbf{S}^{(i+1)} = \langle \mathbf{s'}^1, \mathbf{s'}^2, \ldots, \mathbf{s'}^\kappa \rangle$$

$$p^{(i+1\bullet)} = \langle f(\tau^1), f(\tau^2), \ldots, f(\tau^\kappa) \rangle$$

Figure 6-11: *TimedME* algorithm.

The objective function, f, is the probability of each trajectory from a current system state to a next system state. The probability of a trajectory is computed as the product of the approximate belief state probability $p^{(i\bullet)}[\mathbf{s}^{(i)}]$ associated with the current system state and the probability $\mathbf{P}_\tau(\mathbf{s}^{(i+1)} \mid \mathbf{s}^{(i)}, \mu^{(i)})$ of the transition from the current system state to the next system state. Solving the resulting OCSP using OPSAT (step 4) identifies the leading transitions from the set of possible current system states. OPSAT must also provide the target system states associated with the "optimal" plant transitions, since a plant transition does not capture information about the clock variables.

Once the κ best solutions have been computed, step 5 resets the appropriate clock variables in each solution's target system state. This corresponds to updating the initialization time associated with each component that transitioned into a new mode. Finally, in step 6, *TimedME* returns the κ target system states and their associated probabilities. It should be noted that the conditioning of the probability based on the observation (recall Step 4 in $BeliefUpdate_{TCCA}$) is folded into the OCSP solution step. By including the observation $o^{(i+1)}$ in the OCSP constraint $C(\tau)$, this effectively imposes $\mathbf{P}_{\mathbb{O}} = 0$ or 1, for a refuted or entailed observation, respectively. The case where an observation is neither entailed nor refuted is handled as if the observation were entailed, i.e. $\mathbf{P}_{\mathbb{O}} = 1$. This case results in an overestimation of the actual probability of the system state, because the $1/m$ correction factor has been omitted.

6.3 Mode Reconfiguration

Given the most-likely state estimates computed by ME, MR is used to command the plant to achieve configuration goals issued by the control sequencer. As discussed in Section 4.3.2, the semantics of MR is presented in terms of a variant of decision-theoretic planning on the factored POSMDP plant model. By augmenting the plant state with the clock interpretation, the factored POSMDP is mapped to a factored POMDP, which can then be transformed into a "belief MDP", and solved via dynamic

programming, in theory. In practice, however, the decision theoretic planning problem associated with the belief MDP is known to be intractable [8, 46].

Fortunately, previous work in model-based programming [85] has identified two key assumptions that can be made about the system, which allow the MR problem to be decomposed and solved reactively:

1. *the probability of nominal behavior is assumed to dominate the probability of off-nominal behavior; and*

2. *the reward associated with being in a goal state is assumed to dominate the reward of getting to the goal state.*

Assumption #1 allows MR to disregard fault states and transitions, to focus its planning over nominal states and transitions. Assumption #2 allows reward to be used in determining which of the reachable states corresponds to the "optimal" goal state (i.e., satisfies the configuration goal while maximizing reward) and allows MR to disregard reward in the process of generating the sequence of control actions that leads to the goal state.

Consequently, the implementation of MR is decomposed into two capabilities, the *goal interpreter (GI)* and *reactive planner (RP)*. These capabilities operate in tight coordination with ME, described in Section 6.2. GI uses the plant model and the most likely current state, provided by ME, to determine a reachable goal state that achieves the configuration goal, while maximizing reward. RP takes a goal state and the most likely current state, and generates a command sequence that moves the plant to this goal state. RP generates and executes this sequence one command at a time, using ME to confirm the effects of each command. This decomposition of the MR problem was introduced for the Burton model-based executive [85], and has also been used in the Titan model-based executive [25, 90].

To illustrate the roles of GI and RP, consider an example drawn from the execution of the Mars entry sequence (Chapter 3). Given a configuration goal of *(Engine = Standby)*, the MR operation begins with a call to GI, to determine the maximum-reward (alternatively, least-cost) state in which this goal is achieved. GI finds this goal

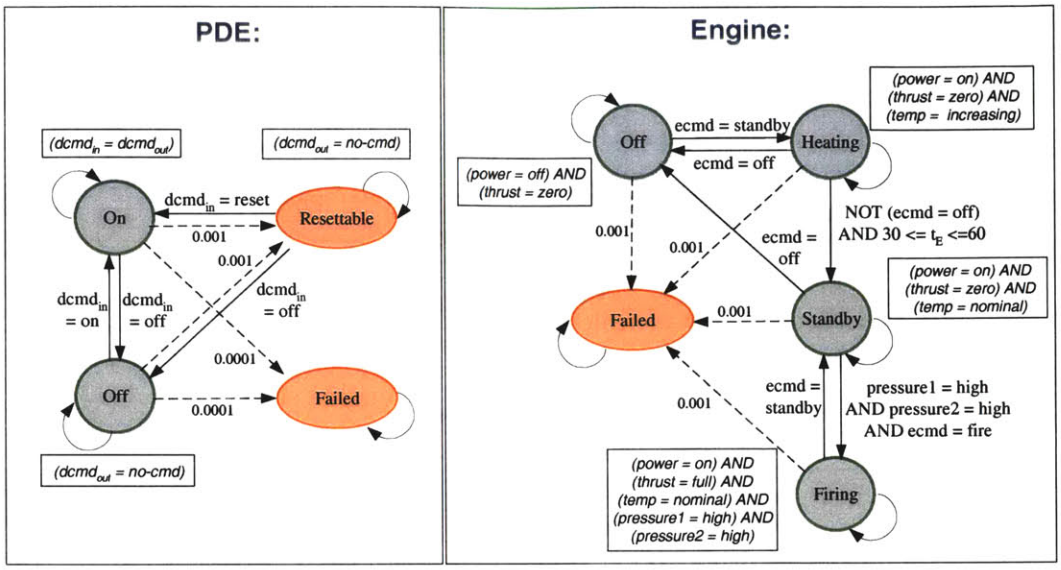

Figure 6-12: Timed Constraint Automata models for the PDE and engine components. The TCCA includes interconnection constraints that link the PDE's output command ($dcmd_{out}$) to the Engine's input command ($ecmd$).

state by reasoning through the TCCA plant model for the system, the relevant subset of which is shown in Figure 6-12. Assuming that a propulsion drive electronics (PDE) unit must be powered on in order to issue commands to the engine, and assuming that the cost in terms of power draw associated with this PDE is non-negligible, the least-cost goal state computed by GI would include the states *(Engine = Standby)* and *(PDE = Off)*. RP then takes this goal state and determines the appropriate ordering of actions that leads to the goal state. In this case, RP issues the sequence of actions "power on PDE" ($dcmd_{in} = on$), followed by "issue engine standby command" ($dcmd_{in} = standbyEngine$). Given confirmation of the *(Engine = Heating)* mode from ME, the correct action for RP is next to wait for the engine to automatically transition into standby mode between 30 and 60 seconds after entering heating mode. Once *(Engine = Standby)* mode is confirmed, RP performs the final action "power off driver" ($dcmd_{in} = off$), to minimize unnecessary power draw. Assuming nominal execution, this sequence results in achievement of the goal state.

As illustrated in the above example, the deductive controller's closed-loop execution capability continues to monitor the progress of the system through to achieve-

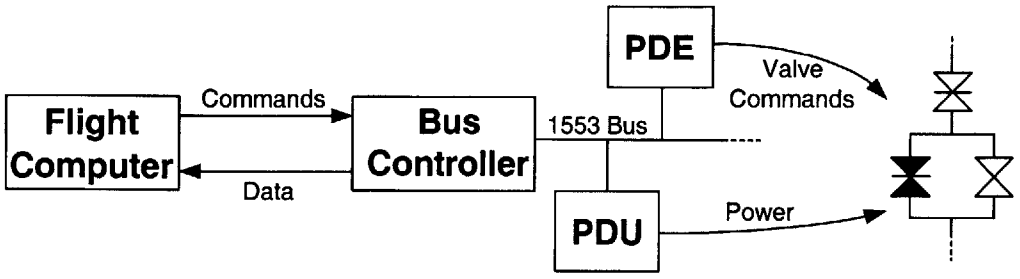

Figure 6-13: A spacecraft has complex paths of interaction: commands issued by the flight computer pass through the bus controller, 1553 bus, and propulsion drive electronics (PDE) on the way to the spacecraft main engine subsystem.

ment of the goal state. This enables an immediate response to off-nominal execution. For example, should the PDE suddenly be determined to have transitioned into its resettable fault mode, RP then issues the appropriate recovery command to ensure achievement of the *(Engine = Standby)* and *(PDE = Off)* goal state. In this case, assuming no further engine commands need to be issued, the appropriate recovery command is $(dcmd_{in} = off)$.

Several properties make the MR problem complex. First, devices are controlled indirectly through physical interactions that are established and removed by changing modes of other devices. Hence, to change a device's mode, a communication path must be established from the control processor to that device, by changing the modes of other devices (see Figure 6-13). Second, because communication paths are shared with multiple devices, mode changes need to be carefully ordered, in order to avoid unintended or destructive effects. Third, since failures occur, the modes of all relevant devices need to be monitored at each step, through all relevant sensors.

The Timed Model-based Executive's MR implementation is based on the MR engine for the Titan model-based executive [90]. Recall that Titan's deductive controller was designed to reason through plant models expressed as untimed Concurrent Constraint Automata (CCA). In this section, a very brief overview of Titan's GI and RP capabilities are provided, followed by a discussion of the issues associated with adapting Titan's MR engine to TCCA plant models and time-critical scenarios.

6.3.1 Overview of Goal Interpretation in Titan

The GI problem is analogous to the ME problem discussed in Section 6.2.3: while ME searches for likely modes that are consistent with the observations, GI searches for modes that entail the configuration goal while maximizing reward [85]. GI generates a maximum-reward plant state $s_g^{(i)}$ that satisfies the current goal configuration $g^{(i)}$ and that is reachable from the current most likely state $\hat{s}^{(i)}$, using nominal transitions whose effects are either *reversible* or correspond to fault repair actions. *Reversible transitions* are defined as transitions for which the source state may be returned to from the target state via a sequence of nominal control actions. *Reversibly reachable states* are states that are reachable from the current state through repeated application of reversible transitions.

Following the analogy with ME, GI can be solved by casting it as an OCSP $\langle \mathbf{x}, f, C \rangle$, as follows. There is a decision variable in $\mathbf{x}$ for each x_j, with domain set to the reversibly reachable modes of component x_j. $C(\mathbf{x})$ is the condition that the target assignment $\mathbf{x}$ is consistent with constraint store C_M, and that $\mathbf{x}$, together with C_M, entail configuration goal $g^{(i)}$. $f = \sum_{(x_j=v)\in s} \mathbb{R}_j(x_j = v)$ is the reward of being in a state s. An example of a typical modal reward metric is power consumption. This OCSP can be solved using OPSAT [91].

6.3.2 Overview of Reactive Planning in Titan

Given a goal state supplied by GI, the RP is responsible for achieving this goal state. Titan's model-based RP is based on the Burton model-based executive [85]. RP takes as input the current most likely state $\hat{s}^{(i)}$ and the goal state $s_g^{(i)}$ selected by GI, and generates the first control action $\mu^{(i)}$ that moves the plant toward the goal state.

RP extends the classic AI planning problem [82] to address the problem of *indirect control*: control variables interact indirectly with internal plant state variables, through co-temporal physical interactions represented in the CCA. RP achieves reactivity by exploiting the requirement that all actions, except repairs, be reversible, and by exploiting certain topological properties of component connectivity that frequently

occur in designed systems. This permits a set of subgoals to be solved serially, i.e., one at a time.

The key to RP's efficiency lies in its offline construction of a set of coupled concurrent policies, one for each state variable, rather than a single global policy for the complete state space. This "divide-and-conquer" approach enables the RP to avoid the state space explosion associated with traditional decision theoretic policies [46] and universal plans [75]. Recent work by Chung [16] has further improved RP's compactness and efficiency by adopting Ordered Binary Decision Diagram representations for the CCA model and the concurrent policies.

Once the concurrent policies have been generated, they are used by RP for online planning. More specifically, given the current state estimate $\hat{s}^{(i)}$ and the goal state $s_g^{(i)}$, RP generates successive control actions $\mu^{(i)}$ as a depth-first traversal through the concurrent policy tables [85]. In the presence of failures, RP generates repair sequences, which are selected so as to minimize the number of irreversible steps taken. The algorithm used by RP to invoke appropriate repair actions is presented in [85].

6.3.3 Extending MR for TCCA and Time-Critical Scenarios

Titan's MR capability (as described in Sections 6.3.1 and 6.3.2) requires some simple augmentations to address two issues that arise from the use of timed plant models, and the focus on time-critical mission scenarios. First, Titan's MR capability was designed to reason through untimed CCA models of plant behavior; what extensions need to be made to the GI and/or RP algorithms to enable them to operate on models with timed transitions? Second, much emphasis was placed on the requirement of control action reversibility (in the nominal execution case); what impact does this have on the execution of the type of time-critical scenario addressed in this thesis? In this section, each of these issues is addressed.

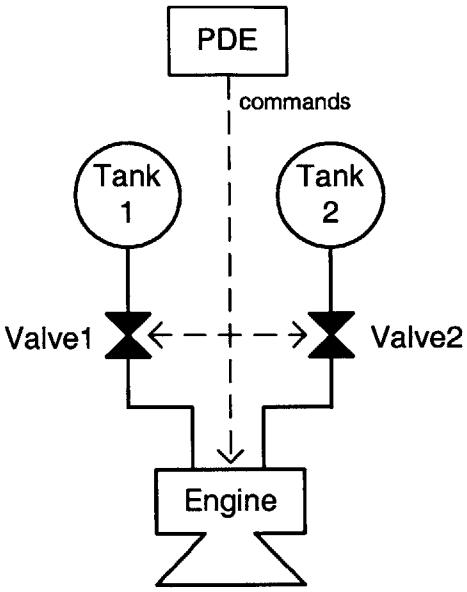

Figure 6-14: Simplified propulsion subsystem for the Mars lander spacecraft.

Addressing the issue of timed plant models

The Titan GI and RP algorithms described in the previous two sections were designed to operate on plant models expressed as CCA, which represent a form of factored POMDP. As described in Section 6.1, this model is inadequate to capture the types of latencies and transient effects that are present in realistic systems. To address this deficiency, the TCCA modeling formalism was introduced in Section 6.1. In this section, a simple extension to Titan's MR algorithms is presented, which allows them to operate on TCCA.

To illustrate the issue, it is helpful to consider an example based on the simplified propulsion subsystem for the Mars lander spacecraft, shown in Figure 6-14. The TCCA models of the PDE, valve and engine components used in this example are shown in Figure 6-15.

Suppose the configuration goal is to reach a state where *(Engine = Firing)*, which is achievable by opening flow of two propellants into the engine and placing the engine in firing mode. Figure 6-16 shows GI's conflict-directed search sequence for the *(Engine = Firing)* configuration goal. The initial state assumes that the PDE is off, both valves are closed, and the engine is off (upper left, Figure 6-16). The desired

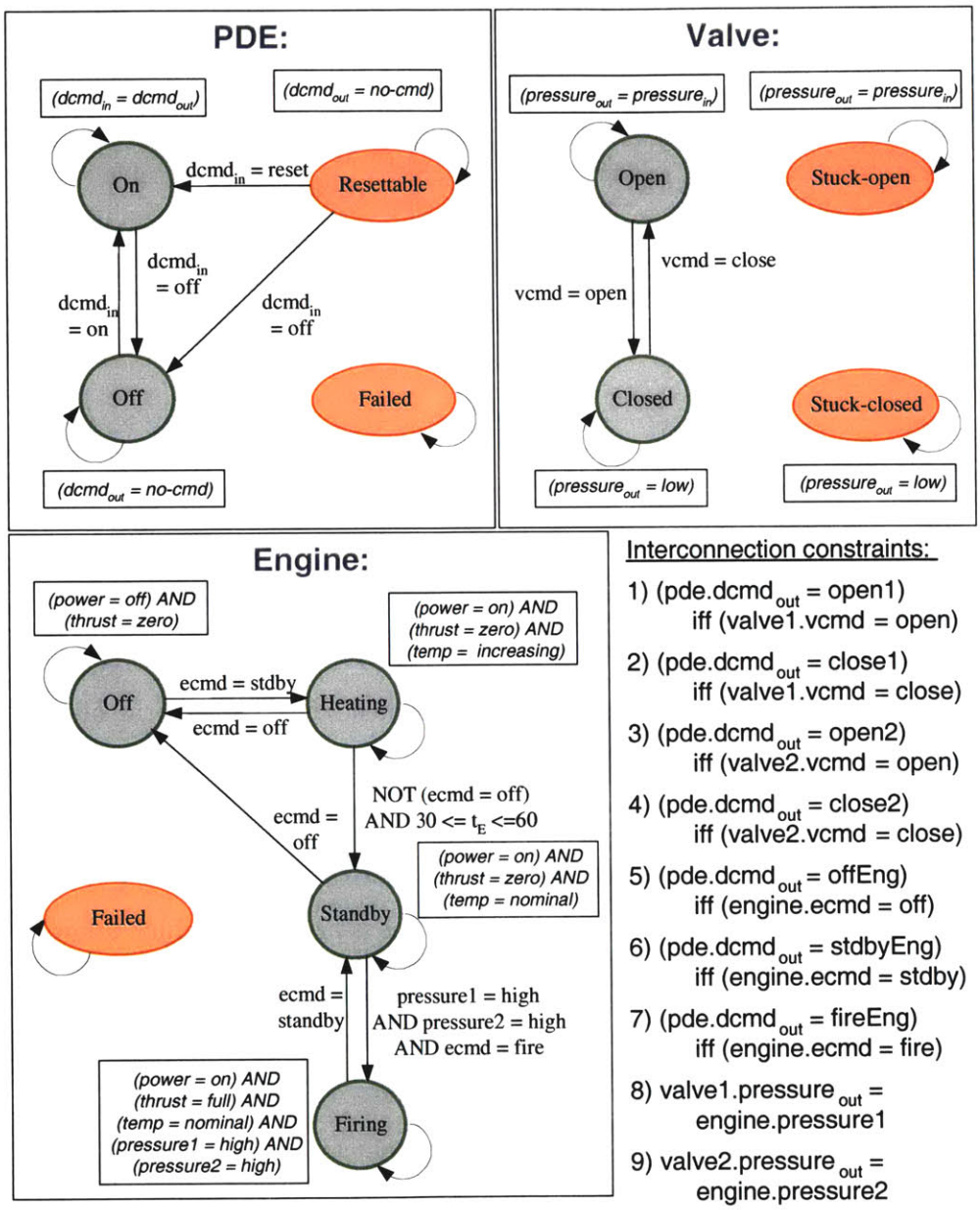

Figure 6-15: TCCA models for a highly simplified propulsion subsystem model. The fault transitions have been omitted for clarity.

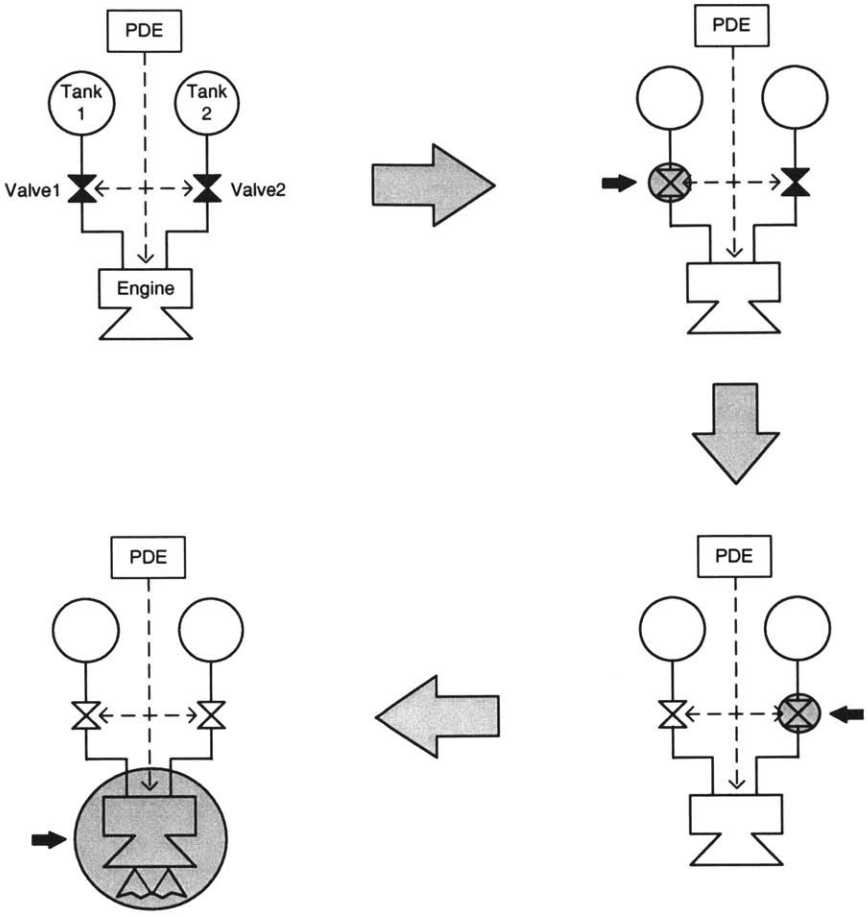

Figure 6-16: Snapshots demonstrating how GI searches for a reachable state that achieves the configuration goal of *(Engine = Firing)* with maximum reward. GI generates the goal state {*PDE = Off, Valve1 = Open, Valve2 = Open, Engine = Firing*} after three iterations, in which it discovers conflicting mode assignments in the tested candidate (highlighted by the small black arrows). The conflict found in each iteration is used to direct the search for the next candidate.

response from the GI capability is that it returns the goal state {*Engine = Firing*, *Valve1 = Open*, *Valve2 = Open*, and *PDE = Off*}. This means that GI must realize that the firing mode is reachable from the initial off mode, despite the existence of an uncommandable timed transition in the path from off to firing. As long as GI's search space, which consists of the cross product of the sets of reachable modes for each component, includes the ultimate goal state, the OPSAT engine will properly consider it for both its conflict-directed search and goal-entailment check operations.

Assuming GI produces the correct goal state, the desired RP behavior is to gen-

Table 6.2: Policy for the engine component.

engine	Target			
Current	*off*	*heating*	*standby*	*firing*
off	**Idle**	*ecmd = standby*	*ecmd = standby*	*ecmd = standby*
heating	*ecmd = off*	**Idle**	**Idle**	**Idle**
standby	*ecmd = off*	*ecmd = off*	**Idle**	*valve1 = open* *valve2 = open* *ecmd = fire*
firing	*ecmd = standby*	*ecmd = standby*	*ecmd = standby*	**Idle**
failed	**Failure**	**Failure**	**Failure**	**Failure**

erate a sequence of control actions that transitions the system from its current state to the goal state. The concurrent policy associated with the engine component is shown in Table 6.2. RP uses this policy to generate the following sequence, assuming nominal execution:

1. issue the standby command to the engine, placing it in heating mode *(Engine.ecmd = standby)*;
2. wait until the engine transitions to standby mode;
3. turn on the driver *(Driver.dcmd$_{in}$ = on)*;
4. open valve 1 *(Driver.dcmd$_{in}$ = open1)*;
5. open valve 2 *(Driver.dcmd$_{in}$ = open2)*;
6. issue the fire command to the engine *(Engine.ecmd = fire)*;
7. turn off the driver to conserve power *(Driver.dcmd$_{in}$ = off)*.

After step 1, the system state is {*Engine = Heating*, *Valve1 = Closed*, *Valve2 = Closed*, and *Driver = Off*}. The RP then realizes that the correct control action to perform is to do nothing at all (i.e., to "wait"): the engine's nominal transition from heating to standby mode should occur automatically from 30 to 60 seconds after step 1. Once ME reports that the standby mode has been reached, the RP proceeds to

steps 3-5, which are necessary to satisfy the two pressure preconditions on the engine's standby-to-fired transition. Upon opening of the second valve (as confirmed by ME), the RP then issues the "fire" command to the engine. After completion of step 6, the system state is {*Engine* = *Firing*, *Valve1* = *Open*, *Valve2* = *Open*, and *Driver* = *On*}. Finally, the only subgoal from the goal state that remains to be satisfied is *Driver* = *Off*}; the RP completes the sequence by issuing the command to turn off the driver.

Now that the desired GI and RP behaviors have been described, it is clear that only a trivial extension to the untimed algorithms is necessary: the set of reversibly reachable modes must be extended to include nominal modes which are reversibly reachable via a timed transition (e.g., the engine's standby mode). During the concurrent policy generation process, time constraints map to the **Idle** control action (as shown in the *heating* $\rightarrow$ *standby* and *heating* $\rightarrow$ *firing* entries in the policy table in Table 6.2. This implies that, in the absence of another control action in the transition guard, the desired control behavior is to "wait" (i.e., repeatedly issue the **Idle** control action) until the time at which the non-deterministic transition is taken.

It should be noted that this implementation of the RP capability has two limitations. First, it does not guarantee that an optimal path to the goal state will be followed (where "optimal" might be defined as involving the least number of control actions, or possibly even defined with respect to some resource metric, such as time, or overall power consumed by the sequence). Second, it enforces serialization of commands, even when no causal link exists between them. This effect is particularly evident in the presence of timed transitions: because RP considers "waiting" for a nominal timed transition to be a sequence of **Idle** actions, and because it treats these **Idle** actions to be control actions like any other command, it will not try to fill this idle time with other control actions that it needs to do, even if no undesired interactions would result. For instance, in the above example, the executive waits for the engine to reach standby mode (step 2) before progressing with the control sequence. In theory, it could further "optimize" the execution of the desired control sequence

by moving ahead to step 3 and turning on the driver while it is waiting.[2]

Addressing the issue of irreversible actions

Chapter 5 showed how the Timed Model-based Executive's control sequencer dictates the execution of the control program based on the most likely state estimate returned by ME. This greedy decision-making approach, which amounts to assuming the most likely estimated state is the correct state, introduces risk: the control action appropriate for the most likely state trajectory may be inappropriate, or worse damaging, if the actual state is something else. Furthermore, the reactive focus of the executive precludes extensive deliberation on the long-term consequences of actions, thus leaving open the possibility that control actions, while not outright harmful, may degrade the system's capabilities. For example, firing a pyro valve is an irreversible action that has a permanent effect on the behavior of the spacecraft's propulsion subsystem. Under non-time-critical circumstances, such actions would generally only be taken after a human operator explicitly reasons through the consequences of the actions over a future time horizon.

The design of the MR capability for the Titan model-based executive ensures the safety of the system by only generating reversible control actions, unless the purpose of the action is to repair failures. While repair actions are irreversible, it is important to allow a reactive executive to repair failures, to ensure robustness. However, within time-critical sequences, the spacecraft does not have the luxury of waiting for a ground operator to effect all irreversible actions that the spacecraft is required to take to successfully complete its mission. For example, the Mars entry scenario presented in Chapter 3 depends on the performance of a critical one-time action, the separation of the lander from the cruise stage. In the full entry, descent and landing scenario described in Chapter 7, a number of such irreversible actions are required, including the deployment of the descent parachute, and the jettison of

[2]It should be noted, however, that opening the valves too far ahead of the fire command would result in an unnecessary loss of propellant. In this case, RP's serialization actually prevents such losses from occurring, because it waits until the engine is in standby mode before opening the two valves.

the heatshield and backshell.

Just as the MR capability was allowed to invoke irreversible control actions to repair component faults, it must be endowed with the ability to perform an irreversible action that is critical to the success of the mission. The Timed Model-based Executive's MR capability must therefore relax the reversibility safety requirement imposed in Titan. Rather than requiring all non-repair actions to be reversible, it allows irreversible actions to be executed if *explicitly* directed to by the control sequencer; that is, if the irreversibly-reachable state is directly specified within the configuration goal. For example, during the execution of the Mars entry sequence described in Chapter 3, when the control sequencer issues the configuration goal *Lander = Separated*, MR overlooks the fact that the *Separated* mode is not reversibly reachable from the *Connected* mode (recall the Timed Constraint Automaton for the *Lander* in Figure 3-3). Thus, MR determines from the model that this goal can be achieved by issuing the *fire-primary* pyro command.

MR can achieve irreversibly-reachable goals which are explicitly requested, provided no irreversibly-reachable intermediate goals are encountered along the way in the reactive planning process. This approach does not achieve irreversible goals that are not explicitly directed, for instance, if an irreversibly reachable component mode appears as an intermediate goal in the concurrent policy for another component. This means that MR will not achieve irreversible goals as "side effects" of other goals. A more detailed discussion of this issue is provided in [16].

6.4 Summary

In this chapter, timed plant models have been specified as TCCA, that is, a composition of Timed Constraint Automata for the components in the system. TCCA provide a constraint-based implementation of the abstract timed plant model described as a factored POSMDP in Section 4.1. TCCA extend the CCA representation adopted in model-based programming [83, 90] by introducing clock variables, clock interpretations, clock initializations, and timing constraints. These time-related augmentations

are based on similar notions from real-time modeling formalisms like Timed Automata [2]. The probabilistic nature of the TCCA model leads to similarities with Probabilistic Timed Automata [52]. TCCA are distinguished from PTA by their constraint-based encoding, and their use of a single clock in each Timed Constraint Automaton. To capture the semi-Markov behaviors of interest in this work, a single clock per component is sufficient.

The deductive controller of the Timed Model-based Executive has been described, in terms of its core ME and MR capabilities. These two capabilities reason through timed plant models encoded as TCCA, to estimate the plant state (ME) and generate an appropriate control sequence that achieves a configuration goal (MR).

The theoretical development of ME presents an algorithm for belief state update for plants modeled as TCCA, based on the definition of system state as the plant state augmented with the current clock interpretations. The intractability of belief state update is addressed in the implementation of ME by framing the problem as an OCSP that can be solved using the OPSAT engine [91]. This approach generates system state estimates in best-first order, while limiting the amount of online search performed and the number of system states tracked.

The Timed Model-based Executive's MR capability builds off the GI and RP algorithms developed for untimed systems [85]. A trivial extension of the notion of "reachability" allows GI and RP to handle plants modeled as TCCA. In addition, an extension to Burton's RP implementation has been presented, to accommodate explicitly-issued irreversible actions, which are key in the execution of mission-critical sequences.

Chapter 7

Executive Implementation and Demonstration

The Timed Model-based Executive has been implemented in C++, as an extension of the Titan model-based executive [90]. This chapter describes how the control sequencer (defined in Chapter 5) and the deductive controller (defined in Chapter 6) are coupled together within the implemented Timed Model-based Execution architecture. It then describes a proof-of-concept demonstration of the Timed Model-based Executive that has been performed in the context of a Mars entry, descent and landing (EDL) scenario.

7.1 Timed Model-based Executive Implementation

The previous two chapters described algorithms for the Timed Model-based Executive's control sequencer and deductive controller modules, and demonstrated their operation individually on simple representative spacecraft examples. This section describes the executive's overall behavior, focusing on how the control sequencer interfaces with the ME and MR modules in a typical execution cycle. Key assumptions made by the Timed Model-based Executive are discussed, and limitations of the executive's implementation are identified.

7.1.1 Execution Architecture

Figure 7-1 provides an interface-level view of the Timed Model-based Execution architecture. The interfaces between the control sequencer and the deductive controller, and between the deductive controller and the underlying physical plant, are provided through Titan's *Real-Time Application Programming Interface (RTAPI).* This RTAPI is derived from that of the Livingstone model-based executive, which was flight demonstrated on the DS-1 spacecraft [5]. The RTAPI "insulates" the deductive controller from the real-time stream of commands and observations arriving from the physical plant, and from the goals and reconfiguration requests arriving from the control sequencer. The RTAPI provides a first-in-first-out queue and a message dispatching process for each of the ME and MR engines, labeled ME_RTAPI and MR_RTAPI in the figure, respectively. When a command or observation arrives from the plant, it is queued up as a message in the ME_RTAPI's incoming queue. The *ME dispatcher* dequeues messages off the queue and makes appropriate ME function calls. Similarly, when a state estimate arrives from the ME dispatcher, or when a goal or MR request arrives from the control sequencer, these messages are pushed onto the MR_RTAPI's queue, for handling by the MR dispatcher.

Interactions with the physical plant are made through the *Control Adapter* and *Monitor Adapter* processes. The Control Adapter's role is to translate the commands received from MR, expressed as assignments to control variables, into executable commands for the actuators. The Monitor Adapter's role is to convert continuous measurement data from onboard sensors into discrete assignments to observable variables, compatible with the timed plant model. In the process of generating the qualitative abstractions of the measurements, the Monitor Adapter process may simply sample the data, compute an average over a short time window, apply some general filtering on the data (including Kalman filtering), or apply feature extraction and symptom-detection operators to discriminate between classes of sensor behavior. This process is analogous to the monitor processes associated with the Remote Agent architecture on DS-1 [5]. A development of the issues associated with monitoring is beyond the

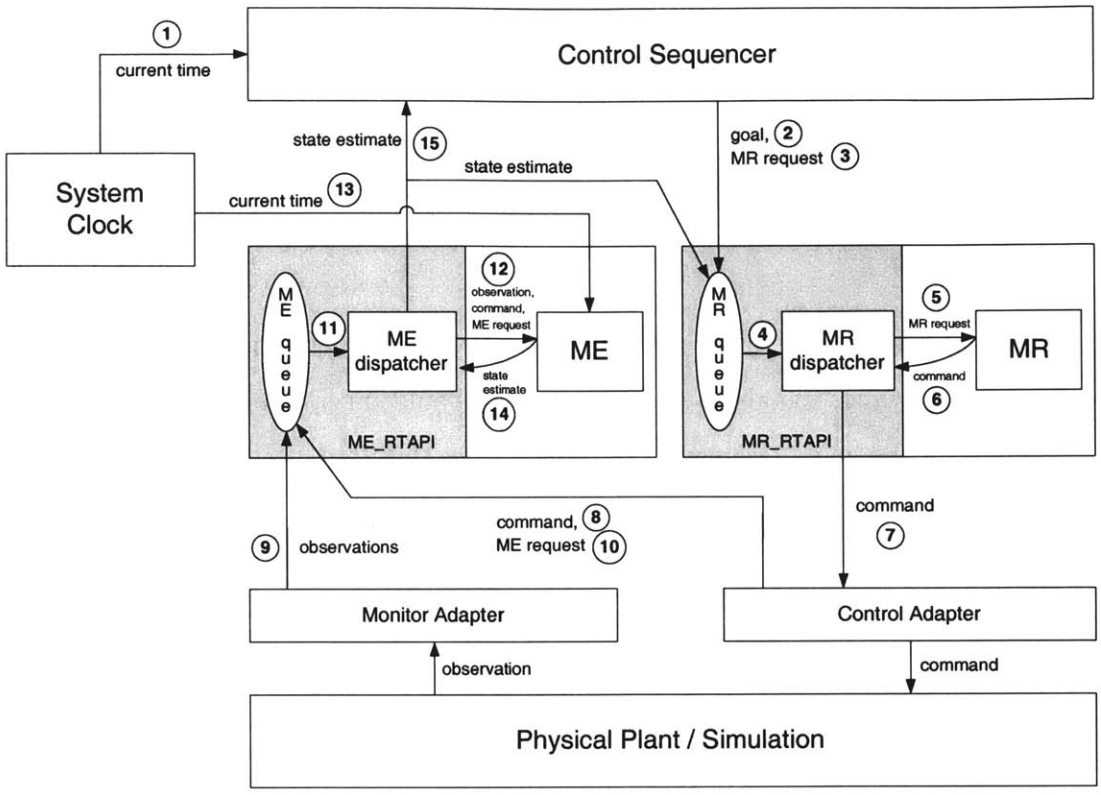

Figure 7-1: Details of the Timed Model-based Execution architecture, based on the Titan model-based executive.

scope of this thesis, but more details can be found in [70, 71]. As discussed below, correct execution of the deductive controller requires that the sample rate of the observations fed into ME by the Monitor Adapter be at least as high as the execution cycle rate (i.e. there is at least one observation per execution cycle).

In order to clearly describe how the integrated executive works, it is helpful to walk through an execution cycle, following the flow of information through the executive architecture. This discussion traces the path of the label numbers in Figure 7-1. The control sequencer and deductive controller are assumed to begin the following cycle with a prior state estimate and clock values:

1. The execution cycle starts with the control sequencer querying the system clock for an update of the current absolute time, which is used to update its clock variables (this corresponds to step 1 in the $Step_{THCA}$ algorithm, defined in Section 5.3). These updated clock values, along with the prior state estimate,

are used to check the maintenance constraints of the currently marked control program locations and to establish which goal constraints in the program are to be asserted in this cycle.

2. The control sequencer asserts these goals by pushing them onto the MR queue, one at a time.

3. Once it has asserted all its goals, the sequencer then queues up an "MR request" message, which signals to the MR that it should initiate an MR computation based on the goals received to date (step 5 in the $Step_{THCA}$ algorithm).

4. As the goal messages arrive in the MR queue, the MR dispatcher dequeues them, and conjoins them into a configuration goal.

5. When the MR dispatcher sees the MR request message, it invokes the MR algorithm by passing it the configuration goal.

6. When MR finishes computing the next command in a sequence that leads to the least-cost state that achieves the configuration goal (as described in Section 6.3), it returns this command to the MR dispatcher.

7. The MR dispatcher sends the resulting command to the Control Adapter, which has the responsibility of issuing the command to the appropriate actuator in the physical plant.

8. The Control Adapter notifies ME that the command has been issued by pushing a command message onto the ME queue.

9. Sensors in the physical plant return measurements that provide evidence of the instantaneous state change resulting from the issued command. A Monitor Adapter translates these measurements into a qualitative observation, which is sent to the ME queue.

10. After a short timeout delay, long enough to allow the "instantaneous" observations to be queued up, the Control Adapter pushes an ME request message

onto the ME queue. For untimed model-based executives, such as Livingstone and Titan, choosing the duration of this timeout delay proves to be a challenge. This is due to their underlying assumption of synchronous Markovian behavior, where the system spends most of its time in a steady state, and the transitions between states are rapid enough that by simply waiting for quiescence (over the short timeout delay), these transitions can be treated as instantaneous. This "waiting for quiescence" step is problematic for many physical plants, in which transitions between states are slow [11]. This requires careful tuning of the timeout delay to balance the need to avoid transients in the observation stream against the need to provide reactivity in the control loop. The additional flexibility of the TCCA modeling framework in terms of capturing timed behaviors allows for explicit modeling of transient states, thus mitigating the problem associated with timeout tuning.

11. As the command and observation messages arrive in the ME queue, the ME dispatcher dequeues them, and invokes the appropriate ME function in order to assert them into the system theory.

12. When the ME dispatcher sees the ME request message, it invokes the ME computation (the *TimedME* algorithm from Section 6.2.3).

13. ME queries the system clock for the current time, so that it can update its plant clock variables (see step 1 of *TimedME*).

14. When ME has finished computing the most likely state estimate (corresponding to an assignment to each component's state variable), it returns this state to the ME dispatcher.

15. Finally, the ME dispatcher sends the state update to the MR engine and the control sequencer (which has been waiting for this new state estimate to be returned, in step 6 of $Step_{THCA}$).

The introduction of time into the execution model adds steps 1 and 13 to the execution cycle for the Titan model-based executive. In step 1, the system time is used

by the control sequencer to advance the current marking of the timed control program, based on clock conditions in the maintenance constraints or transition guards of the THCA (see Section 5.3). In step 13, the system time is used by the deductive controller's ME capability to determine the new most-likely state estimate, as discussed in Section 6.2.3.

7.1.2 Assumptions and Limitations of Implementation

The current implementation of the Timed Model-based Executive makes a number of assumptions to ensure the correct execution of timed model-based programs. This set of assumptions defines a corresponding set of limitations to the executive's capabilities. This section points out two key assumptions made, and associated limitations.

Assumption #1: bounded execution cycle time

Because it involves no search or satisfiability computations, the control sequencer's $Step_{THCA}$ algorithm is worst-case linear in the size of the THCA control program. However, the deductive controller's algorithms, as presented in Chapter 6, are worst-case exponential in the size of the TCCA plant model. This is due to the online OpSat engine's use of search and satisfiability within the control loop. Thus, the length of an execution cycle (i.e., the execution time step resolution) is dictated by the amount of time it takes for ME and MR to complete their computations in each step. To ensure timely execution of control programs, the cycle time of the deductive controller operations must be short.

As it turns out, this assumption is not particularly unreasonable: experience with previous model-based diagnosis engines and executives [20, 50, 83, 85] has shown that only a small fraction of the large state space needs to be explored, to cover essentially all cases that are considered "feasible" by spacecraft systems engineers. Due to the significant cost associated with testing and validation of flight software, typical onboard fault protection is designed to manage (i.e., detect and provide responses for) single faults, or, at worst, two independent faults. Based on a risk analysis and

the fact that spacecraft components are designed to guard against cascading failures, more complex fault scenarios are generally considered so unlikely as to be unnecessary to model and test for. Thus, practically speaking, worst-case performance of the model-based reasoning engine is rarely, if ever experienced: OpSat employs a best-first search to identify candidate states in increasing order of cost (or likelihood, in the case of ME); OpSat's conflict-directed approach rapidly focuses the search space down to a set of candidates that are consistent with the model and observations (in the case of ME) or goals (in the case of GI). The performance of the OpSat engine is discussed in [59, 91].

Nonetheless, current work is underway to address the worst-case performance problem introduced by online model-based reasoning [15, 16, 79, 85]. The idea is to perform the computationally-expensive deduction steps off-line, resulting in compilation of the model into a set of rules that the deductive controller can use to quickly identify most-likely candidate states. In the case of ME, for example, these rules define a mapping from a set of possible observations to a set of possible diagnoses of the component modes [15]. These rules can be fed into an online engine that triggers appropriate rules based on the current observations, and combines the resulting "partial diagnoses" into a most likely state estimate for the system.

The assumption of bounded execution cycle time translates into a limitation on the resolution of time conditions that should be used in the control program. For example, if the executive's cycle time is on the order of 0.1 sec, it makes no sense to specify, in the THCA control program, a transition out of a location that is conditioned on a very small time delay, such as $t1 \geq 0.001$ sec. It should be noted, however, that execution of the timed control program will still proceed correctly, even in the presence of time conditions smaller than the execution cycle time.

Similarly, at the level of the plant models, timed transitions with guards shorter than the executive's time step are considered to occur within a single execution step. For example, consider the timed component model for the driver in Figure 6-4. The transition from the transitional *TurningOn* mode to the *On* mode is conditioned on the time constraint $t_D \leq 0.1$ sec. Unless the execution cycle time is less than 0.1 sec,

the ME computation will integrate the entire probability density function in a single step (that is, the time-dependent probability function shown in row τ_6 of Table 6.1 will reach its final maximum value of 0.989 within a single cycle of the executive). This means that the transition will be taken after one cycle spent in *TurningOn* mode, assuming nominal behavior.

Assumption #2: timely observation updates

The Timed Model-based Executive assumes that observations are received in a timely fashion from the physical plant, in order for them to be useful in discriminating between target states of the non-deterministic timed transitions. For example, consider the example of the engine component from Figure 6-5, where uniform probability density functions over the value of clock t_E are used for the timed transitions. Starting 30 seconds after the engine transitions into *Heating* mode, ME needs to receive observation information, in order to discern whether the transition from *Heating* to *Standby* has been taken, versus the idle transition from *Heating* to *Heating* (and versus an off-nominal transition into *Failed* mode). If no observations are received over the next 15 seconds (i.e., up to $t_E = 45$ sec), ME will determine that the engine most likely transitions into *Standby* in the first execution cycle for which $t_E \geq 45$ sec, because the integrated probability density associated with this transition becomes greater than that associated with the *Heating* to *Heating* idle transition.

7.2 Demonstration

A proof-of-concept demonstration of the Timed Model-based Executive has been performed, by running it through a Mars EDL scenario. This section describes the validation objectives of this demonstration, defines the reference EDL scenario, outlines the set of models (both control programs and plant models) that are used in the scenario, and discusses how the demonstration satisfies the validation objectives.

7.2.1 Testing the Timed Model-based Executive

As shown in Figure 7-1, the Timed Model-based Executive can be interfaced with a simulation environment, which provides a mechanism for injecting realistic observations in response to commands, for a variety of nominal and off-nominal behavior assumptions. The sophistication and complexity of this type of simulator can vary widely, from a simple rule-based simulator, to a stochastic model-based simulator whose models of the simulated plant are at least as detailed as the models used within the executive itself (with probabilities and system parameters that may have been empirically derived, for example). The demonstration described in this section uses a simple simulation framework, developed at the Johns Hopkins University Applied Physics Laboratory, which eavesdrops on the issued command and state estimate streams. Based on the command/state at each execution cycle, the simulator triggers one or more "rules" that inject desired observations some specified number of execution cycles later.

The obvious limitation of this rule-based validation approach (and of case-based testing, in general) is that its scope covers only those off-nominal cases that an application expert has explicitly thought through and provided responses for. Nonetheless, this approach is adequate for demonstrating that the Timed Model-based Executive provides a set of desired capabilities that traditional approaches to encoding critical sequences cannot achieve, or can only achieve at greater cost. The application of more formal verification methods, such as those identified in [61], is identified as an area for future work.

A set of validation requirements has been specified for the Timed Model-based Executive, in the form of key capabilities that the executive must be able to provide. The list of validation requirements has been derived from various sources, including autonomy studies for NASA (such as the New Millenium Program technology roadmap [24]), documentation describing other autonomy software frameworks [49, 69], and documented experience from previous missions (such as DS-1 [5, 60]). Generally speaking, these capabilities can be categorized as either ap-

plicable to nominal operation or operation in the presence of faults. Each of these capabilities is listed below, along with an indication of which module in the executive provides it:

- Validation requirements for nominal operation:

    1. accept high-level activity goals and decompose them (control sequencer)
    2. execute goals conditioned on state constraints (control sequencer)
    3. execute goals conditioned on time constraints (control sequencer)
    4. accept a configuration goal and generate/execute a single-step command that achieves the goal (deductive controller)
    5. accept a configuration goal and generate/execute a multiple-step reconfiguration sequence that achieves the goal (deductive controller)
    6. track nominal modes through commanded transitions (deductive controller)
    7. track nominal modes through timed transitions (deductive controller)

- Validation requirements for operation in the presence of faults:

    1. diagnose faults through commanded transitions (deductive controller)
    2. diagnose faults through timed transitions (deductive controller)
    3. perform recovery by repair (deductive controller, control sequencer)
    4. perform recovery by leveraging physical/functional redundancy (deductive controller, control sequencer)

Though relatively simple, these capabilities cover a large subset of desired features for autonomous spacecraft control. The remainder of this chapter describes a representative mission-critical sequence scenario, presents a set of control programs and models for this scenario, and discusses how each of the above capabilities has been demonstrated in the context of this scenario.

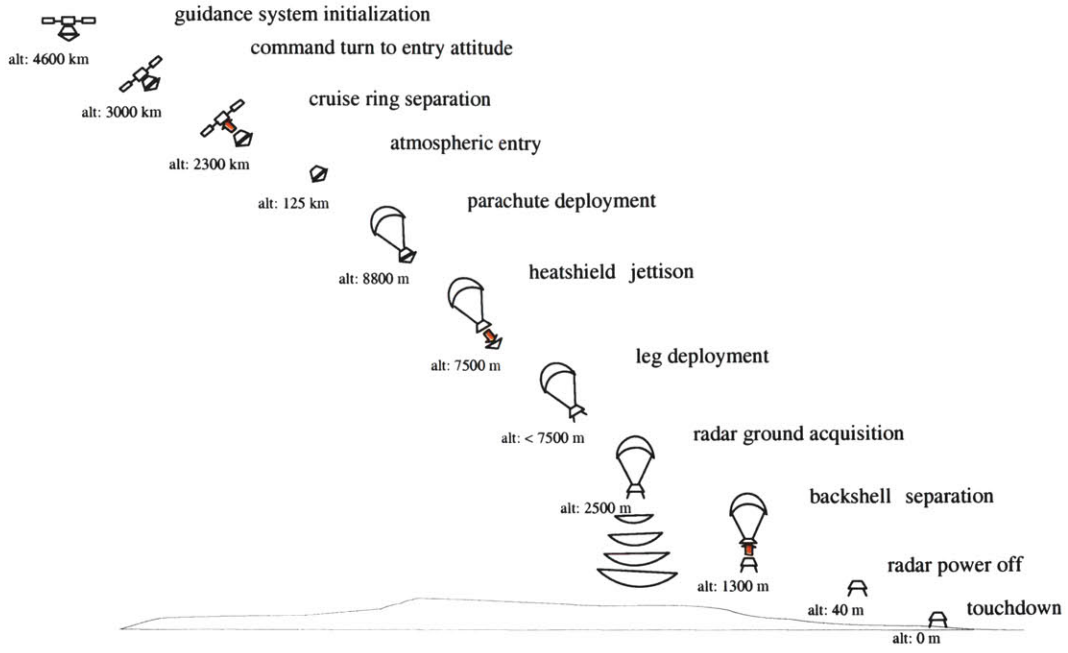

Figure 7-2: Entry, descent and landing sequence for a Mars lander spacecraft.

7.2.2 Mars EDL Scenario Description

Demonstration of the Timed Model-based Executive has been performed in the context of a Mars EDL scenario (see Figure 7-2), which extends the Mars entry sequence presented in Chapter 3 to include the descent and landing phases of the mission. The Mars EDL sequence executes as follows.

At the end of the cruise phase of its mission, as the spacecraft approaches Mars, it turns on and heats up its descent engine, putting it into standby mode. Four and a half hours later, it switches from navigation using a combination of a star tracker and an inertial measurement unit (IMU), to inertial navigation using only the IMU. Four minutes after switching its navigation mode, the spacecraft prepares for atmospheric entry by rotating to its entry orientation. Once the entry orientation has been achieved, the lander stage of the spacecraft separates from the cruise stage and proceeds toward entry into the Martian atmosphere (all the while holding its attitude at the entry orientation).

When atmospheric entry is initiated (as determined by a change in the spacecraft's

acceleration due to atmospheric drag), the spacecraft changes its attitude to zero out its angle of attack relative to its velocity vector. It then holds this attitude as the spacecraft descends into the atmosphere. Once the lander's velocity drops below some threshold value, its parachute is deployed. Ten seconds later (enough time for the spacecraft to stabilize with the deployed chute), the heatshield is jettisoned. The lander's legs are deployed ten seconds after heatshield jettison, to ensure that the deploying legs do not impact the separating heatshield. The landing radar sensor is activated 1.5 seconds later.

Once the radar acquires the ground, providing altitude and descent rate information, the onboard computer computes the time at which the lander's backshell and parachute must be jettisoned. When this time arrives, the backshell is jettisoned, and the descent engines fire 0.5 second later, turning the lander to its proper orientation for landing, and slowing it to its landing speed through a gravity-turn maneuver (in which the thrust vector always pointed in the opposite direction of the instantaneous velocity vector).

The radar is turned off at an altitude of 40 meters above the surface. Once the lander reaches an altitude of 12 meters, it descends vertically to the surface at a constant speed. The descent engines are shut off when touchdown is detected by sensors in the footpads.

In the next two sections, the sets of control programs and plant models that implement this EDL scenario are presented.

7.2.3 Timed Control Programs

Once this type of scenario has been specified, it is fairly straightforward to translate it into a state-based control program, expressed as a THCA. Figures 7-3 to 7-6 depict the THCA control programs for the Mars EDL scenario. The corresponding RMPL code is provided in Appendix B.

In Figure 7-3, the main program for the EDL critical sequence is shown. Upon startup, it initializes two concurrent subautomata, corresponding to the EntrySequence and DescentLandingSequence subprograms. Execution of the overall sequence

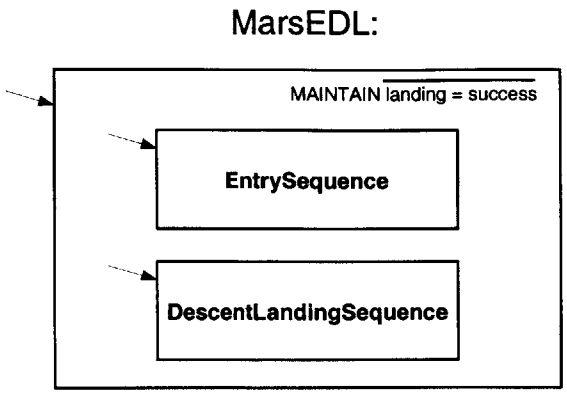

Figure 7-3: Main THCA control program for the Mars EDL sequence. MarsEDL invokes the EntrySequence and DescentLandingSequence control programs in parallel.

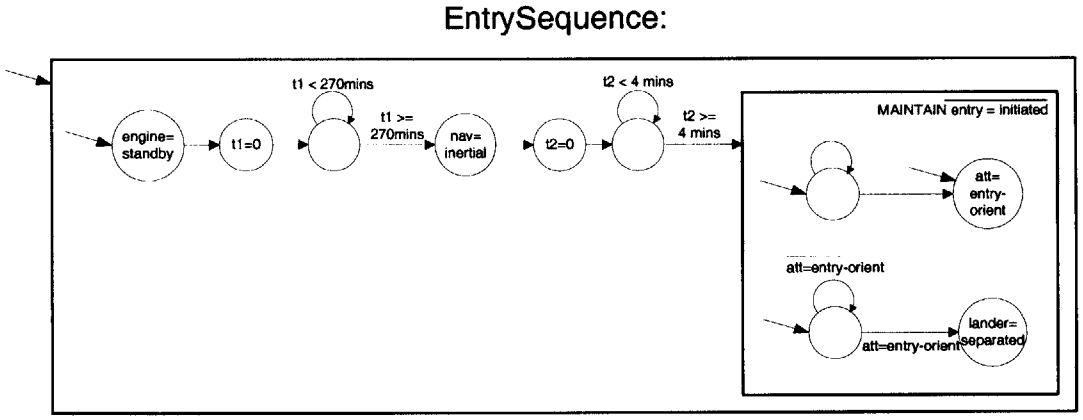

Figure 7-4: THCA control program for the Entry sequence.

completes when the spacecraft lands successfully (corresponding to violation of the maintenance constraint $\overline{landing = success}$).

The EntrySequence control program, which has previously been discussed in Chapter 3, is shown in Figure 7-4. The DescentLandingSequence THCA (Figure 7-5) begins by waiting for the *entry = initiated* state. As soon as this state is entailed, the *lander = separated* goal is reissued. This goal is redundant, having also been issued in the EntrySequence. It is also issued here, in case the EntrySequence was preempted by the premature initiation of Mars entry prior to achieving its *lander = separated* goal.

When the lander is determined to have separated from the cruise stage, the DescentLandingSequence THCA transitions to a composite location with two concurrent

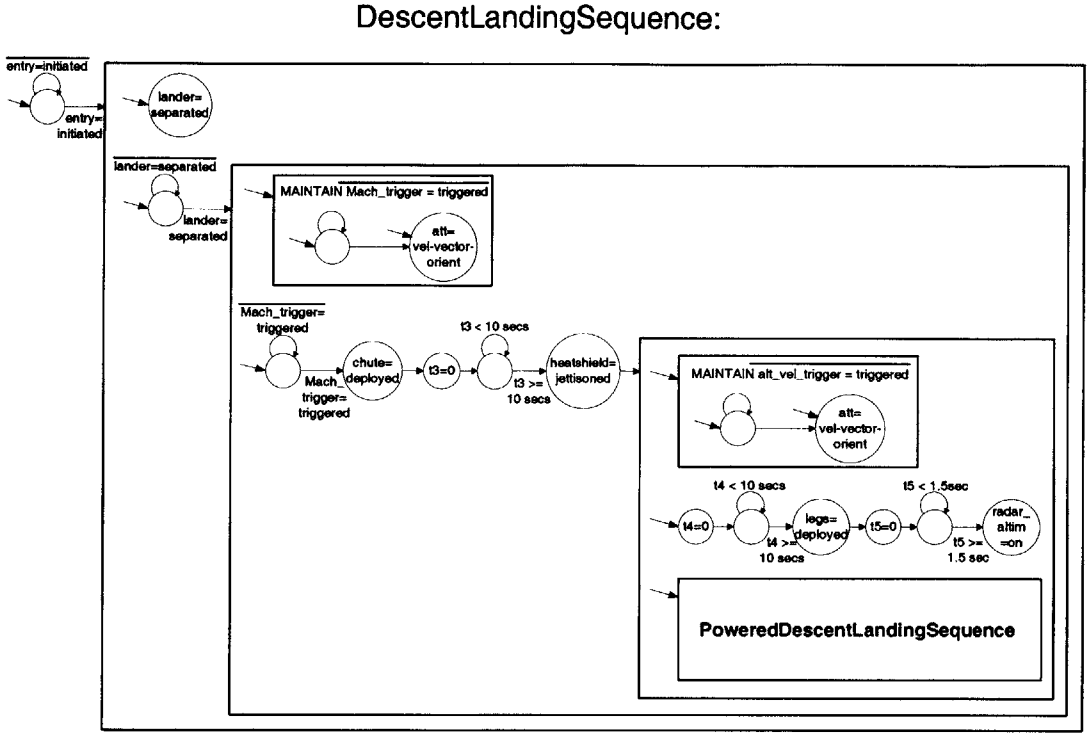

Figure 7-5: THCA control program for the Descent and Landing sequence. DescentLandingSequence invokes the PoweredDescentLandingSequence control program.

threads.[1] The first thread results in the lander holding its attitude in the direction of the velocity vector, until the *Mach_trigger* is determined to be *triggered.* The second concurrent thread waits for this *Mach_trigger* = *triggered* state to be entailed, then proceeds to deploy the parachute. Ten seconds later, the heatshield is jettisoned, and then three concurrent threads are spawned. The first tells the lander attitude control system to resume pointing in the direction of the velocity vector, and to maintain this attitude control mode until the *alt_vel_trigger* is determined to be *triggered.* The second thread delays 10 seconds, deploys the legs, and then 1.5 second later, activates the radar altimeter sensor. The third thread initializes the PoweredDescentLandingSequence THCA, depicted in Figure 7-6.

The PoweredDescentLandingSequence thread waits until the *alt_vel_trigger* = *trig-*

[1]The term "thread" is used here to refer to separate and concurrent activity paths in the THCA control program. It is important to note, however, that this does not imply that the control sequencer process that executes this control program must be multi-threaded; in fact, the current implementation of the control sequencer, as presented in Chapter 5, is single-threaded.

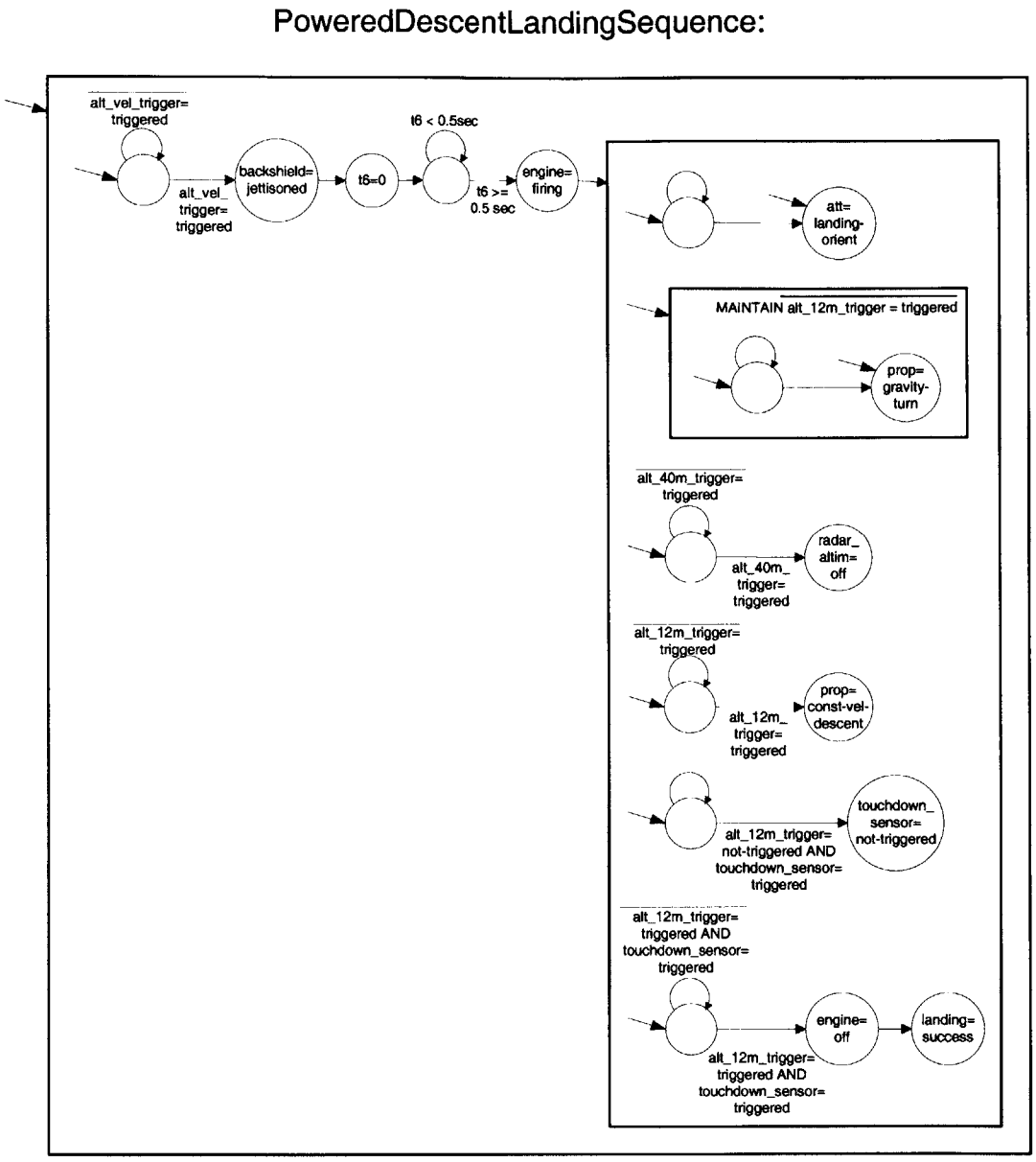

Figure 7-6: THCA control program for the Powered Descent and Landing sequence.

gered before jettisoning the backshield (thereby releasing the lander from the parachute). Powered descent begins 0.5 second later, when the *engine = firing* goal is issued. At this point, six concurrent threads are initiated. In the first thread, the attitude control system rotates the lander and holds it in its landing orientation. In the second thread, the propulsion control system concurrently initiates a gravity turn maneuver. This *prop = gravity-turn* goal is continuously asserted until the *alt_12m_trigger* is

triggered. (3) In the meantime, another parallel thread waits for the *alt_40m_trigger* to be *triggered*, then it turns off the radar altimeter. (4) As soon as *alt_12m_trigger* = *triggered*, the propulsion controller mode is switched to perform a constant-velocity vertical descent from 12 meters altitude. (5) The fifth concurrent thread implements some basic touchdown sensor logic, preventing the sensor from signaling touchdown before the 12-meter altitude trigger is engaged. This basic logic protects against a recurrence of the type of fault that is presumed to have led to the demise of the Mars Polar Lander spacecraft, as described in Chapter 1. (6) Finally, one more concurrent activity waits for the simultaneous entailment of both *alt_12m_trigger* = *triggered* and *touchdown_sensor* = *triggered*, at which point it turns off the engine and signals *landing* = *success*. As mentioned above, this results in successful termination of the overall MarsEDL THCA.

7.2.4 Timed Plant Models

This section provides an overview of the various spacecraft component models built for the Mars lander spacecraft example. Since the goal of this demonstration is to show that the Timed Model-based Executive satisfies the set of validation requirements in Section 7.2.1, the plant models implemented here are far fewer and generally simpler than those one would expect to assemble for a real spacecraft. In particular, many details of the spacecraft are abstracted away for the purpose of this demonstration, including important functions such as power distribution through the system and bus communication between the processor and devices. Consequently, a number of the "components" described below actually correspond to abstracted models of multiple physical components, or sometimes entire subsystems. By implementing these models, the assumption is made that lower-level controllers are folded into the plant, to interpret the abstract states and control the lower-level hardware appropriately. It should be noted that this paradigm of abstraction enables the Timed Model-based Executive to operate as a "configuration manager", responsible for commanding switches between lower-level controller modes. This higher-level role is appropriate for the Timed Model-based Executive in its interactions with spacecraft subsystems like the Atti-

tude Control Subsystem (ACS), which closes very tight high-frequency control loops around the spacecraft's attitude sensors and pulse-frequency modulated thrusters, for example. It would be impossible to insert the action of Timed Model-based Executive into these control loops and still maintain the necessary level of control bandwidth.

The implemented plant models focus on the states that are referenced in the EDL control program's goal constraints, maintenance constraints, and transition guards. The set of TCCA plant models also includes additional components that are coupled to the referenced components, to demonstrate the executive's ability to manage more complex low-level interactions and reason about hidden state. The components/states modeled for the EDL demonstration scenario are listed in Table 7.1.

This set of models includes several classes of state variables:

- **hardware component states**, such as the propulsion subsystem components (*engine, PDE, valves, tanks*), the onboard sensors (*radar_altim, touchdown_sensor*), and the various pyro components (*lander, chute, heatshield, legs, backshield*);
- **discrete abstractions of continuous states**, such as *att*;
- **controller/estimator mode states**, such as *prop* and *nav*; and
- **event triggers**, such as *entry, Mach_trigger, alt_vel_trigger, alt_40m_trigger, alt_12m_trigger,* and *landing.*

In the remainder of this section, discussion is focused on the TCCA corresponding to a subset of these state variables, representing each of the four classes. The full set of TCCA for the demonstration scenario is included in Appendix C.

Hardware component states

As an example of the modeled hardware components, the propulsion subsystem of the Mars lander spacecraft is considered. Figure 7-7 illustrates the interconnections between the six components that make up the subsystem. This is an abstraction of an actual spacecraft propulsion subsystem, made up of two tanks that feed propellant

Table 7.1: List of TCCA plant models for the Mars EDL demonstration example.

State Variable	Model Description	Type
engine	spacecraft main engine	hardware component
PDE	propulsion drive electronics	hardware component
valve1	propulsion system valve	hardware component
valve2	propulsion system valve	hardware component
tank1	propellant tank	hardware component
tank2	propellant tank	hardware component
nav	navigation estimator mode	controller/estimator mode
entry	spacecraft's entry trigger state	event flag
att	spacecraft attitude state	continuous state abstraction
lander	lander separation pyro subsystem	hardware component
Mach_trigger	Mach number trigger state	event flag
chute	parachute pyro subsystem	hardware component
heatshield	heatshield pyro subsystem	hardware component
alt_vel_trigger	altitude/velocity trigger state	event flag
legs	lander leg subsystem	hardware component
radar_altim	rader altimeter sensor	hardware component
backshield	backshield pyro subsystem	hardware component
prop	propulsion controller state	controller/estimator mode
alt_40m_trigger	40-meter altitude trigger state	event flag
alt_12m_trigger	12-meter altitude trigger state	event flag
touchdown_sensor	touchdown sensor state	hardware component
landing	landing trigger state	event flag

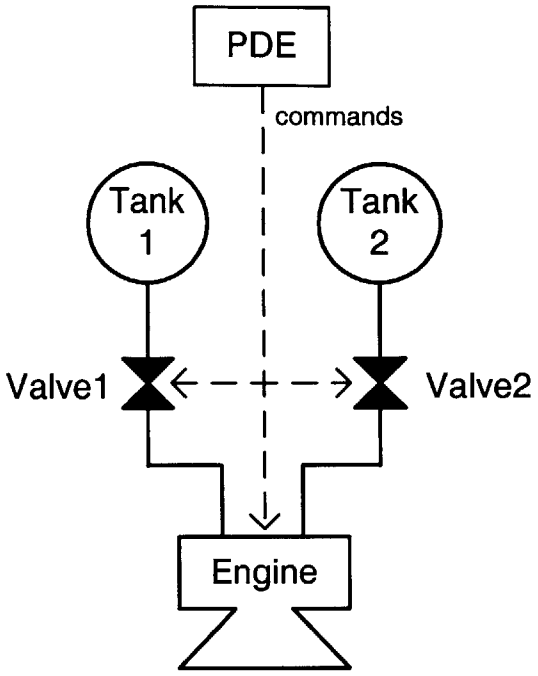

Figure 7-7: Simplified propulsion subsystem for the Mars EDL demonstration scenario.

to the spacecraft's main engine (in this case, the usually complex path from tank to engine is simplified to a single valve component). The valves and engine are commanded through the PDE. The only input to this system is the PDE command, and the following observations are available: the pressures downstream of either valve, the power input and engine temperature of the engine, and the spacecraft thrust level.

The *tank* component is modeled as having two possible modes, a nominal mode *full* and a fault mode *empty*. A full tank provides high pressure at its outlet, and an empty tank provides low pressure. The *valve* component model is shown in Figure 7-8. The valve has nominal modes *open* and *closed*, and fault modes *stuck-open* and *stuck-closed*. The modal constraints are shown in boxes adjacent to the corresponding modes.

The *engine* model, shown in Figure 7-9, has been previously introduced in Section 6.3.3. Recall that the engine exhibits timed behavior: it nominally spends between 30 and 60 seconds in heating mode before the transitioning to standby mode, based on the engine temperature having reached its *nominal* level. The probability

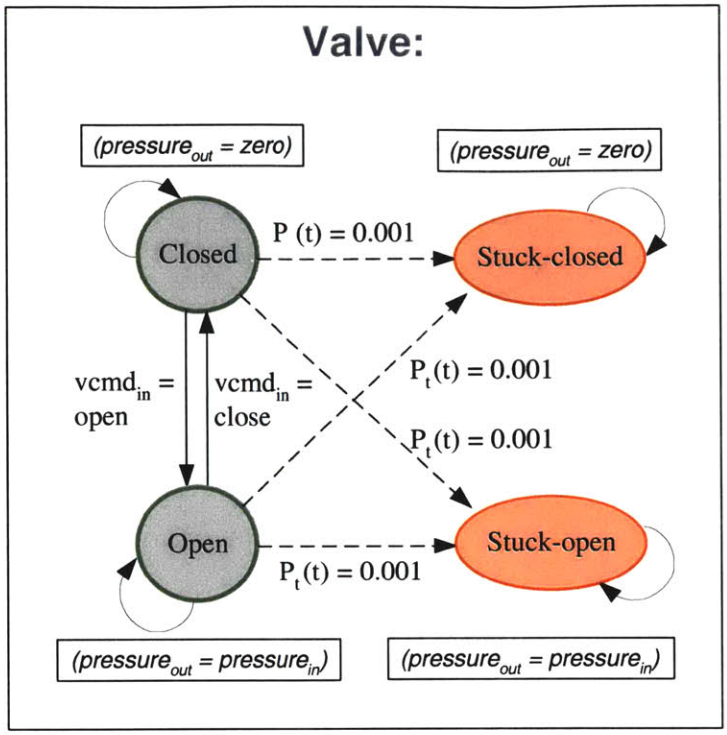

Figure 7-8: Timed Constraint Automaton model for the *valve* component.

density function associated with this timed transition is assumed to be uniform from 30 to 60 seconds, and zero outside this interval. It should also be noted that the *firing* mode corresponds to an operational mode where the propulsion controller has the authority to throttle the engine as necessary. This throttling process is not captured in the engine plant model, but is handled within the real-time control layer that sits beneath the deductive controller.

The simplified model of the propulsion drive electronics was previously presented in Figure 6-15. When the *PDE* is in its *on* mode, commands that are sent to it are passed along to the connected components (in this case, the two valves and the engine). A *resettable* fault mode is specified, which can be repaired by issuing a *reset* or *off* command to the PDE. An unconstrained and permanent *failed* mode is also specified.

TCCA models for the other hardware components are included in Appendix C.

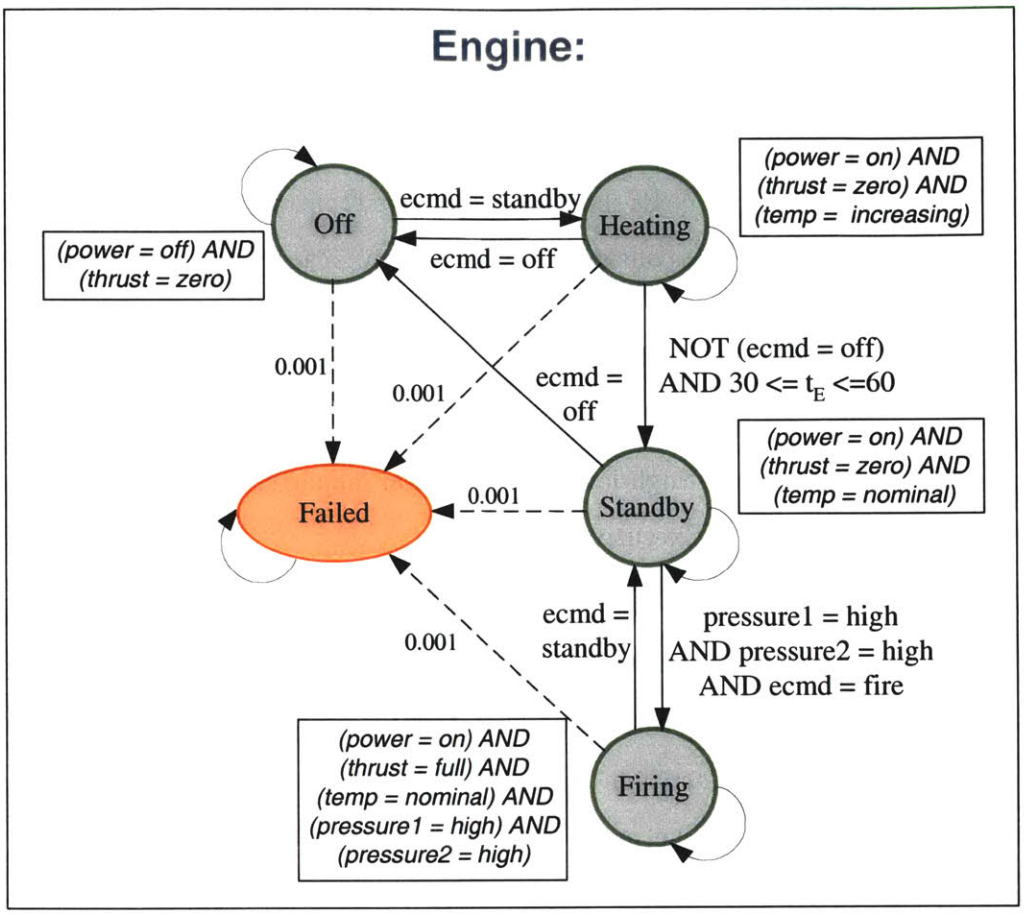

Figure 7-9: Timed Constraint Automaton model for the *engine* component.

Discrete abstractions of continuous states

Many spacecraft states, such as its attitude, are generally described using continuous representations, such as three dimensional vectors or quaternions. In order to be able to reason about these states at the level of the Timed Model-based Executive's deductive controller, discretized abstractions of these states must be considered. For example, the TCCA model for the *att* state variable provides a discretized abstraction of the spacecraft's continuous attitude state. This model has been discretized by defining modes corresponding to the attitude set-points of interest at the level of the control specification (namely, *cruise-orient*, *entry-orient*, *vel-vector-orient*, and *landing-orient*), and transitional modes corresponding to the act of slewing between these set-points. This model is inherently qualitative - it is assumed that the underly-

ing attitude control system knows what specific attitude quaternions each qualitative description (e.g., *entry-orient*) corresponds to, at any given time.

The utility of this type of qualitative representation is that it allows the control sequencer to specify attitude goals that map to specific "control modes" of the attitude control system. This idea is consistent with the way spacecraft systems engineers generally define trajectories, at the level of the control sequences: it is generally more informative to specify a goal of the form *att = landing-orient*, and to allow the attitude control system to work out the details of what specific pointing angle this corresponds to. This type of high-level configuration management role is well suited for the Timed Model-based Executive.

The *att* model is shown in Figure 7-10. The modal constraints make reference to the *att-obs* observable, which can be thought of as qualitative feedback from the attitude control system; e.g., the attitude control system will return *att-obs = in-entry-orient* if the current attitude quaternion is within some small range around the entry-orient set-point. The modal constraints also refer to assignments to the *ACS* variable, which captures the current control mode of the attitude controller. This value is assumed to be returned by the underlying ACS as an observable. As modeled, the *att* TCCA transitions from any set-point mode to the slewing modes associated with all other set-points (most of the transitions are "stubbed-out" in the figure, for the sake of clarity).[2] The time bounds on the transitions from a slewing mode to its corresponding set-point mode are set conservatively to $0 \leq t_A \leq 300$sec, to allow plenty of time for a full range of spacecraft rotation in each slew; that is, the control authority of the ACS actuators is assumed to be sufficient to easily rotate the spacecraft from any arbitrary attitude to *entry-orient* within 300 seconds. For the purposes of this demonstration, uniform probability density functions over time are assumed for each timed transition.

[2]Though all these transitions are possible, the desired trajectory specified by the control program follows a linear path through these modes: *cruise-orient* → *slew-to-entry* → *entry-orient* → *slew-to-vel-vector* → *vel-vector-orient* → *slew-to-landing* → *landing-orient*.

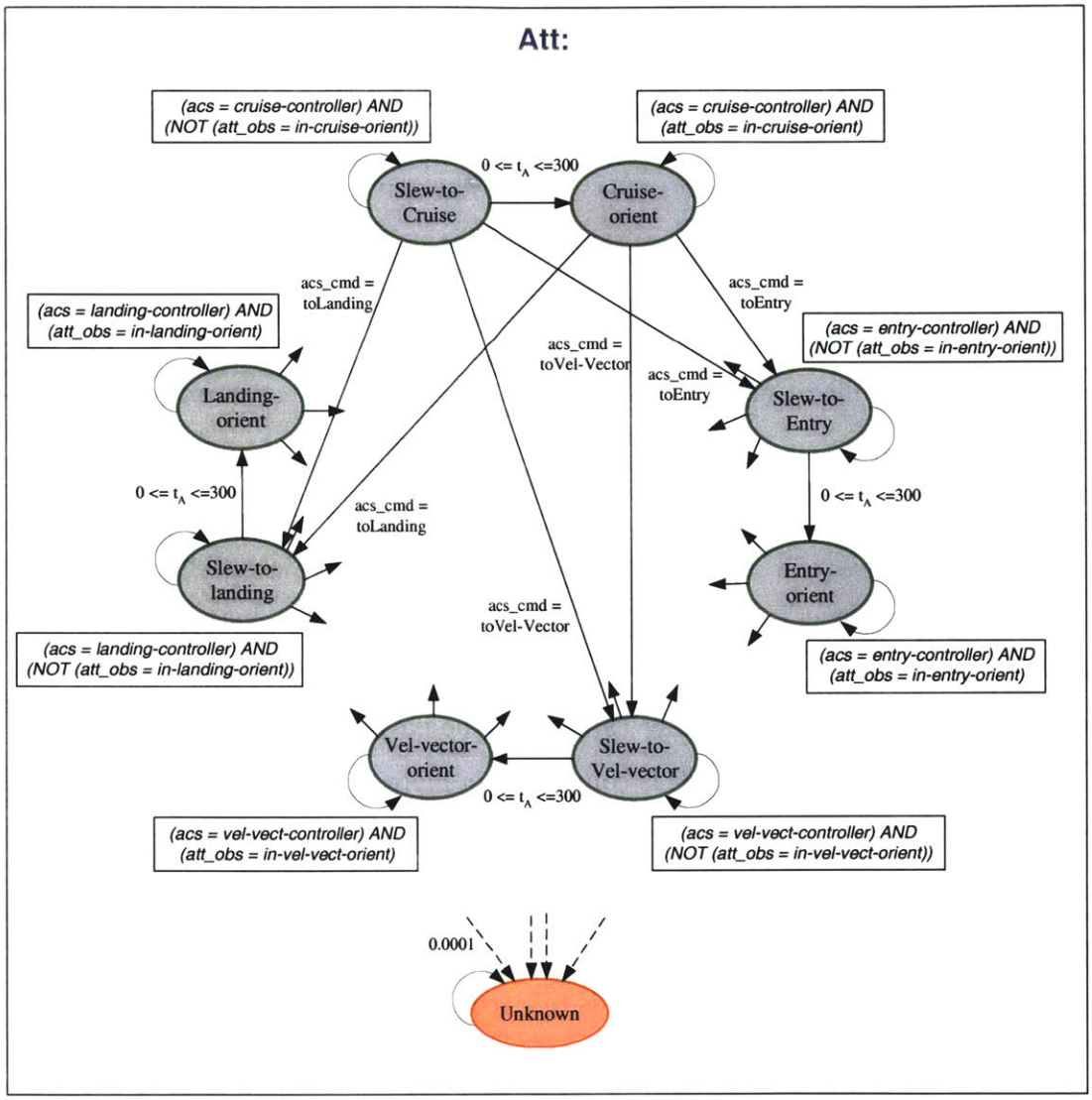

Figure 7-10: Timed Constraint Automaton model for the *att* state. Not all the possible transitions are shown.

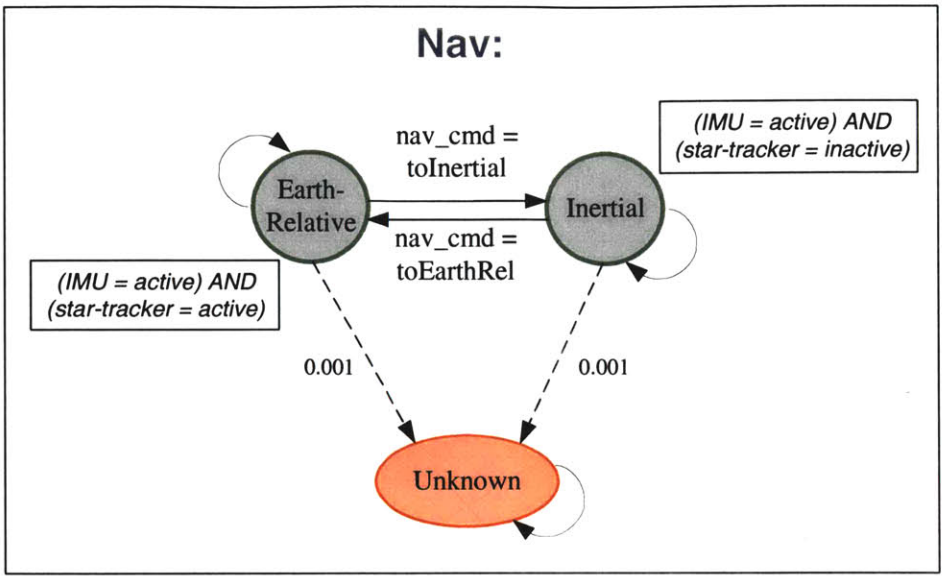

Figure 7-11: Timed Constraint Automaton model for the *nav* estimator state.

Controller/estimator modes

As described above, the Timed Model-based Executive is well-suited to act as a "configuration manager", by commanding appropriate switches between lower-level controller or estimator modes. In this scenario, the executive serves in this role with respect to the propulsion controller (*prop*) and the navigation estimator (*nav*). The *nav* model, shown in Figure 7-11, is considered in more detail here.

During the cruise stage of the mission, the navigation estimator on the spacecraft uses all sensors at its disposal (in this case, both an IMU and a star tracker) to compute its position relative to the Earth, as accurately as possible. This mode of operation for the navigation estimator is referred to as *Earth-relative* mode. Prior to the spacecraft entering the Mars atmosphere, it disables its star tracker, switching over to *inertial* navigation. Though faults in software components are not generally considered, the model specifies an unconstrained *unknown* fault mode.

Event triggers

The fourth type of state variable present in the demonstration model is event triggers. Event triggers correspond to software "flags" that are set as a result of observables

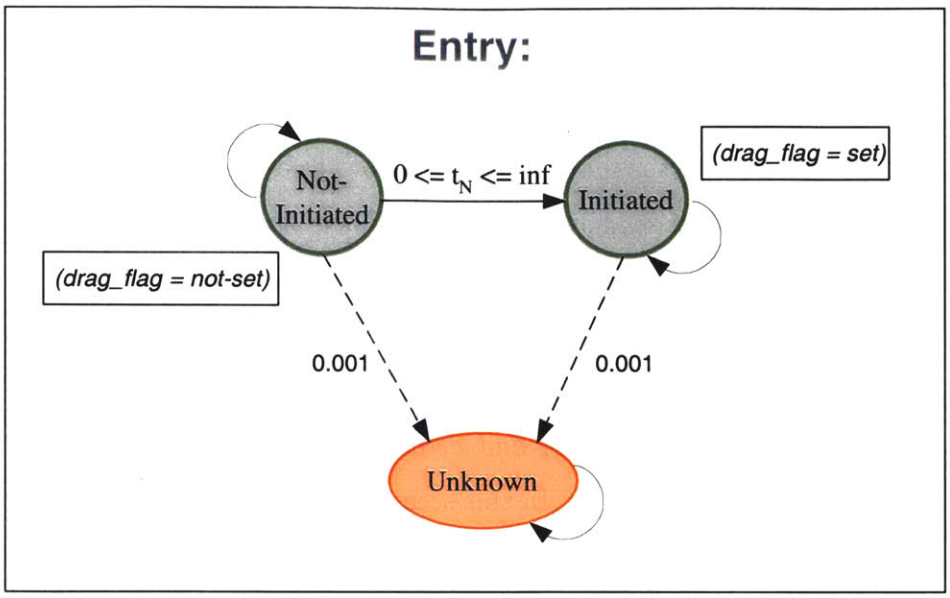

Figure 7-12: Timed Constraint Automaton model for the *entry* state trigger.

in the plant model. They provide a mechanism for execution of a control program to be conditioned on observed information. Figure 7-12 depicts the model for the *entry* state trigger, which is tied to the detection of atmospheric drag by onboard accelerometers. The transition from *not-initiated* to *initiated* is timed but unbounded (its upper bound is infinite), and the probability density function is a gaussian with mean equal to the predicted time of atmospheric entry, based on pre-launch trajectory calculations.

This completes the discussion of the plant models for the Mars EDL demonstration example. In the following section, a brief discussion of how the execution of the Mars EDL control program in conjunction with these models addresses the validation requirements identified in Section 7.2.1.

7.2.5 Validated Capabilities

Now that the demonstration scenario has been presented, and its control programs and plant models described, it remains to discuss how each of the aforementioned validation requirements are satisfied in this scenario, for both the nominal and off-nominal capabilities.

Nominal operation

Successful monitoring and execution of nominal spacecraft operations are clearly critical, as one typically expects the majority of operations to be conducted in the absence of faults. This section discusses how the Timed Model-based Executive provides the set of nominal operation capabilities mentioned above. First, the executive must be able to *accept high-level activity goals* or procedure invocations (either generated onboard using a system-level planner/scheduler, such as ASPEN [14] or EUROPA [45], or uplinked from ground controllers) *and decompose them* into an appropriate sequence of detailed task assignments. In this scenario, the onboard planner or ground controller would simply issue the "initiate MarsEDL" activity goal to the Timed Model-based Executive. This goal would invoke the MarsEDL control program. The executive's control sequencer demonstrates the ability to decompose this high-level activity goal into a valid series of subactivities (e.g., EntrySequence, DescentLandingSequence) and, further, into configuration goals for the deductive controller.

Second, the executive must be able to execute sequences conditioned on state and time constraints. The demonstration scenario clearly highlights this capability, as its execution depends on spacecraft state (e.g., attitude), a number of "external" event triggers, and clock conditions.

Third, the executive must be able to take a configuration goal and generate an appropriate sequence of atomic plant commands that will achieve this goal. More specifically, the executive should be able to generate both single-step and multi-step reconfiguration sequences, as required. The Mars EDL demonstration highlights the ability of the Timed Model-based Executive's deductive controller to do this. Simple goals, such as *nav* = *inertial*, are reachable with a single plant command. Multi-step nominal reconfigurations are demonstrated in the achievement of goals like *engine* = *firing*, which requires the executive to deduce that the engine must be heated, the PDE must be turned on, the valves must be opened and finally, the engine must be commanded to fire (see detailed discussion in Section 6.3.3).

Finally, the executive must be able to perform nominal mode tracking through

both commanded and timed transitions. This capability is key to the proper execution of control programs, as a thread of execution (i.e., a marked location in the THCA) does not progress until the deductive controller confirms that the specified goal has been satisfied. In the Mars EDL scenario, for instance, *pressure = high* measurements from the propulsion subsystem are used to confirm the belief that a valve has successfully opened when commanded to do so, during achievement of the *engine = firing* goal. Similarly, receipt of a *temp = nominal* observation allows the executive to conclude that the engine has taken the timed transition from *heating* to *standby* mode.

Off-nominal operation

While the general expectation is for the bulk of spacecraft operations to be conducted under nominal operating conditions, the ability to autonomously detect and correct/manage faults is essential, especially in environments where timely ground intervention is impossible. Thus, there exist a number of important capabilities related to autonomous operation in the presence of faults.

One core requirement is the ability to diagnose faults, both as a result of unsuccessful commanded transitions, and unsuccessful timed transitions, as well. The Mars EDL scenario allows for demonstration of fault diagnoses in several ways, including detection of an unsuccessful pyro firing for the *lander* separation, and detection of a fault in the *engine* component as a result of it spending too much time in its *heating* mode, without reaching its nominal temperature.

Another key consideration with respect to fault diagnosis is the management of completely unanticipated faults. The Timed Model-based Executive provides a mechanism for accommodating such faults, through the definition of unconstrained fault modes in the TCCA plant models (frequently referred to as "unknown" modes in the plant models). This unconstrained state definition provides runtime robustness by serving as a fallback for component failure modes not explicitly encoded by the spacecraft engineer. This feature can be demonstrated in various ways in the Mars EDL scenario, including the case where the *engine* component suddenly observes a

loss of power while in its heating mode.

Once faults have been detected, the executive must be able to take necessary action to return to nominal operations, if possible. One expectation is that the executive should be able to effect recovery by repairing faulty components. This is demonstrated, for example, by the ability to issue a *reset* command to the PDE component should it experience a resettable fault. As directly repairing a faulty component is not always possible, an executive must also be able to take advantage of physical or functional redundancy to address faults. In the scenario, physical redundancy in the *lander* pyro component allows for proper separation of the lander stage from the cruise stage, after an *unsuccessful-attempt* due to misfire of the primary pyro mechanism.

In conclusion, the Mars EDL scenario allows the Timed Model-based Executive to demonstrate each of the key capabilities, both nominal and off-nominal. Further testing of the executive with more complex models and control programs is planned, in the context of ongoing research in infusing TMBP technology into actual flight missions.

Chapter 8

Conclusions

The work described in this thesis has been motivated by the need to improve the robustness and reduce the development costs of traditional approaches to encoding mission-critical spacecraft sequences. The TMBP paradigm provides specifications of timed control programs and plant models, which are expressive enough to capture the *time-dependent* behaviors that characterize the state evolution of complex asynchronous systems, such as spacecraft. The timed control programs correspond to specifications of desired system state trajectories, leveraging this *state-based* abstraction to reduce the complexity inherent in traditional programming approaches, which interact with a physical plant in terms of low-level sensor measurements and actuator commands. Furthermore, the *visual* nature of the control and plant specifications make them more appealing to the systems engineers in charge of designing the mission, and ensuring the survival of the spacecraft through its critical sequences.

This thesis has defined a Timed Model-based Execution framework that allows for *direct execution* of these state-based control specifications by employing a deductive controller that performs in-the-loop automated reasoning through the plant models. By providing the deductive controller with models of both nominal and off-nominal component behavior, the Timed Model-based Executive is inherently *fault-aware*; that is, it diagnoses and responds to faults on-the-fly, within the nominal state monitoring and achievement loop. In addition, Timed Model-based Execution offers the following advantages, which can result in decreased software development costs as compared to

traditional approaches to complex sequence execution:

- **Modularity** - Because of its use of modular system models, flight software written as a model-based program can accommodate component-level modifications late in the spacecraft design cycle. New component models can be swapped in without having to rewrite significant sections of flight code. Furthermore, modularity within the deductive layer allows for transparent upgrading of the engines used for mode estimation and mode reconfiguration, when more powerful ones become available.

- **Model Reusability** - Another benefit to the use of modular system models is that the component-level models can be reused. Over time, a database of models for different subsystems and component designs can be assembled, dramatically reducing the need for single-use flight code.

- **Verifiability** - As a consequence of being able to write control code in terms of states, model-based programming results in cleaner code that is easier to verify. Ease of verification is also a feature of the plant models, which can be built up directly from system engineering specifications of hardware or software components. It should be noted that the strong influence of formal approaches to real-time modeling means that model-based programs are amenable to formal verification via model-checking tools, such as SPIN [41], KRONOS [10] or UPPAAL [55]. This is identified in Section 8.1 as a promising area for future research.

It is important to re-emphasize the fact that the model-based programming framework defines a family of languages, each characterized by a choice of underlying plant modeling formalism. For example, [90] introduces a model-based programming language that defines a plant model in terms of a factored untimed POMDP. This language and its corresponding executive has been demonstrated through deployments on a wide range of applications from the automotive and aerospace domains [25]. This thesis defines a more general instance of a model-based programming language based

on a factored timed POSMDP representation of a physical plant, which subsumes the untimed language of [90]. The practical importance of this instance has been demonstrated in the context of mission-critical scenarios, such as Mars EDL.

In addition, model-based programming allows for a family of executives for each language, with each executive characterized by the implementation details of its control sequencer and deductive controller modules. For instance, several variants of deductive controller have been documented in [50, 83, 85]. The capabilities of these variants differ by the nature of the approximations made in their respective model-based reasoning algorithms; for example, the Livingstone2 [50] ME capability extends the coverage of Livingstone's ME [83] to track multiple summarized histories of likely past states, while the Burton [85] model-based executive extends Livingstone's one-step MR capability to generate a sequence of control actions that lead to a desired goal state. Another variant of deductive controller currently under development includes compiled versions of mode estimation [79] and mode reconfiguration [16], which improve the performance of the online reasoning by moving the most computationally intensive steps off-line, via pre-compilation of the plant model. A variant of control sequencer is found in the Kirk executive [48, 81], which plans and executes a course of action for networks of robots by reasoning through a non-deterministic control program that encodes alternative time-bounded activities.

In the remainder of this chapter, directions for future work are suggested and the thesis contributions are summarized.

8.1 Directions for Future Work

Many areas have been identified as potentially fruitful targets for future research. In this section, a few of the most interesting possible extensions of this work are highlighted, from both the theory and implementation perspectives:

- **Formal Verification of Timed Model-based Programs** - As discussed in Chapter 4, the semantics of TMBP borrows various ideas from the semantic descriptions of real-time specification languages, such as Timed Automata [2]

and Timed Transition Systems [39]. Such languages are characterized by their amenability to verification via formal tools. Key ideas from these real-time specification languages are also folded directly into the languages used to implement timed model-based programs, at both the control program and plant model levels (see Chapters 5 and 6).

The strong influence of formal approaches means that model-based programs should be similarly amenable to formal verification via model-checking tools, such as SPIN [41], KRONOS [10], UPPAAL [55] or SMV [12]. In fact, work has already been done on the formal verification of untimed Livingstone plant models [61, 63], and the Remote Agent's procedural executive [38]. Porting these tools to the types of models used in TMBP (namely, timed control programs expressed as THCA and timed plant models expressed as TCCA) would enable better characterizations of the coverage of complex spacecraft plant models (via reachability analysis of states in the TCCA) and various important safety properties (such as the identification of potential negative interactions between concurrently-asserted goals in the THCA control program).

- **Hybrid Model-based Programming** - The Mars EDL example discussed in this thesis includes states, such as spacecraft attitude and altitude, that would be more accurately modeled as continuous, rather than discretized qualitatively as in the timed model-based programs. Current work is underway to extend the TMBP framework to a full *hybrid model-based programming* paradigm, where goals and conditions can be expressed in terms of both continuous and discrete variables. This extension of TMBP will define yet another language in the family of model-based programming languages. In a hybrid model-based executive, the deductive controller will reason through hybrid models composed of both discrete behavior models and continuous state dynamics described by ordinary differential equations. This effort will unify the recent work done in hybrid mode estimation [40] with the sophisticated discrete inference capabilities of the Timed Model-based Executive's deductive controller.

One primary thrust of this effort will be to transparently integrate, into the deductive controller, state-of-the-art continuous estimators and control engines that are currently deployed onboard spacecraft. By first focusing on the integration of widely deployed estimators and controllers, the hybrid model-based programming approach will target the "comfort zone" of spacecraft systems engineers, and thus provide an incremental, but nonetheless very significant, improvement in robust onboard executive capability.

- **Visual Design Tools** - In order to further TMBP's stated goal of making the systems engineer's job easier, a suite of visual tools should be assembled to facilitate the design and analysis of THCA control programs and TCCA plant models. These tools should be designed with an intuitive graphical user interface, in the spirit of *Stanley* and the *Model-based Skunkworks* tools [68] developed at NASA Ames for use with the Livingstone2 model-based executive.

  Preliminary work in this area has been initiated at the Johns Hopkins University Applied Physics Laboratory, where a plant visualization tool called *Helios* has been developed and interfaced with the Titan model-based executive (see Figure 8-1). Helios is a Java application that provides the capability to visualize the ME and MR portions of the deductive controller both graphically, in the form of a schematic display, and textually, in tabular format. Another tool for visualizing the execution of Titan's control programs has been developed, based on the Generic Modeling Environment (GME2000) toolkit [56] from Vanderbilt University.

- **Deductive Controller Improvements** - The deductive controller implementation presented in this thesis has a number of limitations, as discussed in Chapter 6. Future research should be done to develop a second-generation deductive controller for the Timed Model-based Executive. Improvements should be made to both the ME and MR capabilities. The second-generation ME should provide a better approximation to the "exact" belief state update process, for example, by enabling ME to track a set of most likely trajectories over a

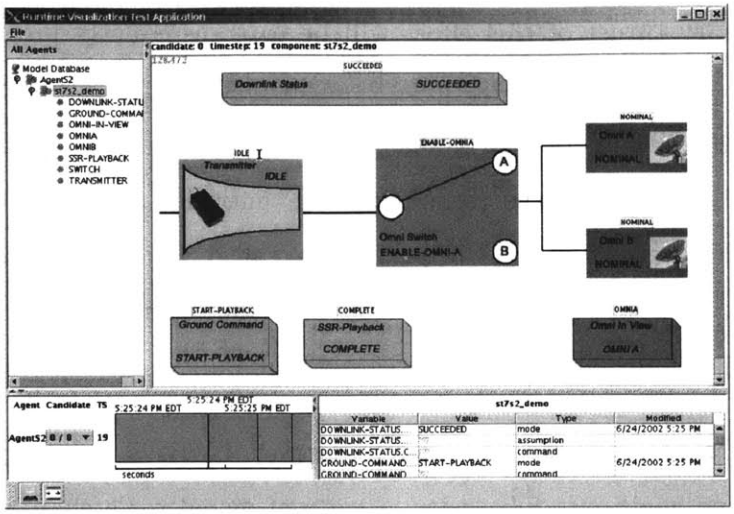

Figure 8-1: Screen snapshot from the Helios visualization tool.

time history, similar to the Livingstone2 capability [50], but operating on timed plant models of the type described in Chapter 6 (TCCA). The MR capability could be improved by choosing a more complex optimality criterion than the current "maximum state reward" objective function, for example, by including in the cost metric the time required to reach a state goal.

Finally, both deductive controller modules should be upgraded with algorithms that enable them to utilize models where the computationally expensive on-line model-based reasoning operations have been "pre-compiled" into a set of model-derived rules, which can be used to diagnose or command the spacecraft in time that is linear in the number of rules. Such compiled versions of ME and MR have already been developed for the untimed deductive controller employed by the Titan model-based executive [15, 16, 79].

8.2 Summary of Contributions

The work documented in this thesis has three key contributions:

1. **Semantic Specification for TMBP** - The semantics of the timed model-based programming approach have been detailed. A timed control program is

modeled as a deterministic automaton that operates on the plant state. The plant model is represented as a factored Partially Observable Semi-Markov Decision Process that captures both nominal and off-nominal behavior, and is encoded compactly using concurrency and constraints. The execution semantics of the timed model-based program are defined in terms of timed sequences of control program locations and legal state evolutions of the physical plant POSMDP. The semantics of the Timed Model-based Executive modules have also been specified. The control sequencer executes a timed control program and issues configuration goals for achievement by the deductive controller, which operates on the factored POSMDP model of the physical plant. The deductive controller provides a mode estimation capability, which performs a variant of belief state update, and a mode reconfiguration capability, which performs a variant of decision theoretic planning. The deductive controller semantics leverage the key insight that the plant state can be augmented with the plant clock interpretations, leading to a mapping from a semi-Markov to a Markov process.

2. **Definition of graphical and textual TMBP languages** - Graphical and textual languages for encoding timed control programs and plant models have been specified. The textual RMPL language provides standard constructs for expressing reactive control behavior, such as conditional branching, iteration, parallel composition, sequential ordering and preemption. The design of RMPL unifies key ideas from synchronous programming, constraint programming, and robotic execution languages. Graphical languages are used to encode both control programs (THCA) and plant models (TCCA). The adoption of a visual encoding for timed model-based programs allows them to be specified and readily inspected by the systems engineers in charge of designing mission-critical sequences.

3. **Development of a Timed Model-based Executive** - A Timed Model-based Executive is defined as consisting of two components, a *control sequencer* and

a *deductive controller*. The control sequencer executes a timed control program conditioned on time and state constraints, by issuing state-based configuration goals. The deductive controller is responsible for estimating the plant's most likely current state based on observations from the plant, and for issuing commands to move the plant through a sequence of states that achieve the configuration goals. Key features of the control sequencer include its implementation of closed-loop goal-driven execution, and its consideration of time via a set of active clock variables. The deductive controller operates on timed (semi-Markov) plant models. Its mode estimation capability implements an approximate algorithm for belief update of the *system state*, where system state is defined as a set of assignments to state variables, augmented by assignments to plant clock variables that keep track of the amount of time each component has spent in its current mode. The mode reconfiguration capability extends the untimed model-based executive's MR capability, by addressing issues associated with timed plant models and irreversible actions.

Bibliography

[1] M. Abadi and L. Lamport. An old-fashioned recipe for real-time. In *Proceedings of the REX Workshop on Real-Time: Theory in Practice*, volume 600 of *Lecture Notes in Computer Science*, pages 1–27. Springer-Verlag, 1991.

[2] R. Alur and D.L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[3] R. Alur. Timed automata. In *11th International Conference on Computer-Aided Verification*, volume 1633 of *Lecture Notes in Computer Science*, pages 8–22. Springer-Verlag, 1999.

[4] D. Bernard et al. Design of the Remote Agent Experiment for spacecraft autonomy. In *Proceedings of the IEEE Aerospace Conference*, 1999.

[5] D. Bernard et al. Final report on the Remote Agent Experiement. In *NMP DS-1 Technology Validation Symposium*, February 2000.

[6] G. Berry and G. Gonthier. The synchronous programming language Esterel: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[7] G. Berry. The Esterel v5 Language Primer, version 5.21 release 2.0. Technical report, Centre de Mathematiques Appliquees, Ecole des Mines and INRIA, April 6 1999.

[8] D. Bertsekas. *Dynamic Programming and Optimal Control.* Athena Scientific, 1995.

[9] C. Boutilier and D. Poole. Computing optimal policies for partially observable decision processes using compact representations. In *Proceedings of AAAI-96*. AAAI Press, 1996.

[10] M. Bozga et al. Kronos: A model-checking tool for real-time systems. In *Proc. 10th International Conference on Computer Aided Verification*, volume 1427, pages 546–550. Springer-Verlag, 1998.

[11] J. Bresina, K. Golden, D. Smith, and R. Washington. Increased flexibility and robustness of Mars rovers. In *Proceedings of ISAIRAS-99*, 1999.

[12] J. Burch et al. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, June 1992.

[13] J. Casani et al. Report on the loss of the Mars Polar Lander and Deep Space 2 missions. Technical report, JPL Caltech, March 2000.

[14] S. Chien et al. Using iterative repair to increase the responsiveness of planning and scheduling for autonomous spacecraft. In *Proceedings of AIPS-00*, 2000.

[15] S. Chung, J. Van Eepoel, and B.C. Williams. Improving model-based mode estimation through offline compilation. In *Proceedings of ISAIRAS-01*, 2001.

[16] S. Chung. Decomposed symbolic approach to reactive planning. S.M. thesis, Massachusetts Institute of Technology, Department of Aeronautics and Astronautics, Cambridge, MA, 2003.

[17] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. *Journal of the ACM*, 5(7), 1962.

[18] R. Davis. Diagnostic reasoning based on structure and behavior. *Artificial Intelligence*, 24:347–410, 1984.

[19] J. de Kleer and B.C. Williams. Diagnosing multiple faults. *Artificial Intelligence*, 32(1):97–130, 1987.

[20] J. de Kleer and B.C. Williams. Diagnosis with behavioral modes. In *Proceedings of IJCAI-89*, pages 1324–1330, 1989.

[21] A. Di Pierro and H. Wiklicky. An operational semantics for probabilistic concurrent constraint programming. In *Proceedings of the International Conference on Computer Languages (ICCL 98)*, pages 174–183. IEEE Computer Society Digital Library, 1998.

[22] D. Dvorak et al. Software architecture themes in JPL's Mission Data System. In *Proceedings of the AIAA Guidance, Navigation, and Control Conference*, Portland, OR, 1999.

[23] D. Dvorak. Challenging encapsulation in the design of high-risk control systems. In *Proceedings of the 17th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA-02)*, 2002.

[24] L. Fesq et al. Spacecraft autonomy in the new millenium. In *Proceedings of the 19th Advances in the Astronautical Sciences (AAS) Guidance and Control Conference*, number AAS-96-001, February 1996.

[25] L. Fesq et al. Model-based autonomy for the next generation of robotic spacecraft. In *Proceedings of the 53rd International Astronautical Congress of the International Astronautical Federation (IAC-02)*, number IAC-02-U.5.04, October 2002.

[26] R.J. Firby. *Adaptive Execution in Dynamic Domains.* PhD thesis, Yale University, Department of Computer Science, 1989.

[27] E. Gat. ESL: A language for supporting robust plan execution in embedded autonomous agents. In *Plan Execution: Problems and Issues, Papers from the 1996 AAAI Fall Symposium*, pages 59–64, 1996.

[28] E. Gat and B. Pell. Smart executives for autonomous spacecraft. *IEEE Intelligent Systems*, 13(5):56–61, Sep./Oct. 1998.

[29] E. Gat. The MDS autonomous control architecture. In *Proceedings of the Fourth World Automation Congress (WAC), International Symposium on Intelligent Automation and Control (ISIAC-2000)*, number ISIAC-062, 2000.

[30] C. Goodrich and J. Kurien. Continuous measurements and quantitative constraints - challenge problems for discrete modeling techniques. In *Proceedings of ISAIRAS-01*, 2001.

[31] P. Le Guernic, M. Le Borgne, T. Gauthier, and C. Le Maire. Programming real time applications with Signal. *Proceedings of the IEEE*, 79(9):1321–1336, September 1991.

[32] C. Guestrin, D. Koller, and R. Parr. Solving factored pomdps with linear value functions. In *Proc. of IJCAI-01 Workshop on Planning under Uncertainty and Incomplete Infromation*, pages 67–75, 2001.

[33] V. Gupta, R. Jagadeesan, and V. Saraswat. Models of concurrent constraint programming. In *Proceedings of the International Conference on Concurrency Theory (CONCUR 96)*, volume 1119 of *Lecture Notes in Computer Science*, pages 66–83. Springer-Verlag, 1996.

[34] N. Halbwachs, P. Caspi, and D. Pilaud. The synchronous programming language Lustre. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[35] N. Halbwachs. *Synchronous Programming of Reactive Systems.* Series in Engineering and Computer Science. Kluwer Academic, 1993.

[36] E. Hansen. An improved policy iteration algorithm for partially observable mdps. In *Proceedings of the Conference on Neural Information Processing Systems*, pages 1015–1021, 1997.

[37] D. Harel. Statecharts: A visual formulation for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.

[38] K. Havelund et al. Formal analysis of the remote agent before and after flight. In *Proceedings of 5th NASA Langley Formal Methods Workshop*, Williamsburg, VA, June 2000.

[39] T.A. Henzinger, Z. Manna, and A. Pnueli. Timed transition systems. In *Proceedings of the REX Workshop on Real-Time: Theory in Practice*, volume 600 of *Lecture Notes in Computer Science*, pages 226–251. Springer-Verlag, 1992.

[40] M. Hofbaur and B.C. Williams. Mode estimation of probabilistic hybrid systems. In *Proceedings of the International Conference on Hybrid Systems: Computation and Control*, May 2002.

[41] G. Holzmann. The model checker spin. *IEEE Trans. on Software Engineering*, 23(5):279–295, 1997.

[42] R. Howard. *Dynamic Programming and Markov Processes.* MIT Press, Cambridge, MA, 1960.

[43] M. Ingham, R. Ragno, and B.C. Williams. A reactive model-based programming language for robotic space explorers. In *Proceedings of ISAIRAS-01*, 2001.

[44] M. Ingham et al. Autonomous sequencing and model-based fault protection for space interferometry. In *Proceedings of ISAIRAS-01*, 2001.

[45] A.K. Jonsson et al. Planning in interplanetary space: Theory and practice. In *Proceedings of AIPS-00*, 2000.

[46] L. Kaelbling, M. Littman, and A. Cassandra. Planning and acting in partially observable stochastic domains. *Artificial Intelligence*, 101, 1998.

[47] Y. Kesten and A. Pnueli. Timed and hybrid statecharts and their textual representation. In *2nd International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 571 of *Lecture Notes in Computer Science*, pages 591–619. Springer-Verlag, 1992.

[48] P. Kim, B.C. Williams, and M. Abramson. Executing reactive, model-based programs through graph-based temporal planning. In *Proceedings of IJCAI-01*, volume 1, pages 487–493, 2001.

[49] K. Kolcio, M. Hanson, and L. Fesq. Validation of autonomous fault diagnostic software. In *Proceedings of the IEEE Aerospace Conference*, volume 4, pages 251–264, 1998.

[50] J. Kurien and P. Nayak. Back to the future for consistency-based trajectory tracking. In *Proceedings of AAAI-00*, pages 370–377, 2000.

[51] J. Kurien, P. Nayak and D. Smith. Fragment-based conformant planning. In *Proceedings of AIPS 2002*, 2002.

[52] M. Kwiatkowska, G. Norman, R. Segala, and J. Sproston. Verifying quantitative properties of continuous probabilistic timed automata. In *Proceedings of the International Conference on Concurrency Theory (CONCUR 2000)*, volume 1877 of *Lecture Notes in Computer Science*, pages 123–137. Springer-Verlag, 2000.

[53] C. Largouet and M. Cordier. Adding probabilities to timed automata to improve landcover classification. In *ECSQARU-2001 Workshop on Spatio-Temporal Reasoning and Geographic Information Systems*, 2001.

[54] C. Largouet and M. Cordier. Using model-checking techniques for diagnosing discrete-event systems. In *Proceedings of DX-01*, pages 39–46, 2001.

[55] K. Larsen, P. Pettersson, and W. Yi. UPPAAL in a nutshell. *Int. Journal on Software Tools for Technology Transfer*, 1(1–2):134–152, 1997.

[56] A. Ledeczi et al. The generic modeling environment. In *Proceedings of IEEE International Workshop on Intelligent Signal Processing (WISP-2001)*, Budapest, Hungary, May 2001.

[57] J. Lunze. Diagnosis of quantized systems based on a timed discrete-event model. *IEEE Transactions on Systems, Man, and Cybernetics – Part A: Systems and Humans*, 30(3), May 2000.

[58] S. Mahadevan. Partially observable semi-markov decision processes: Theory and applications in engineering and cognitive science. In *Proceedings of AAAI Fall Symposium on Planning with Partially Observable Markov Decision Processes*, 1998.

[59] P. Nayak and B. C. Williams. Fast context switching in real-time propositional reasoning. In *Proceedings of AAAI-97*, pages 50–56, 1997.

[60] P. Nayak et al. Validating the DS1 Remote Agent Experiment. In *Proceedings of ISAIRAS-99*, 1999.

[61] S. Nelson and C. Pecheur. Formal verification of a next-generation space shuttle. In *Proceedings of the Second Goddard Workshop on Formal Aspects of Agent-Based Systems (FAABS II)*, Greenbelt, MD, October 2002.

[62] C. Papadimitriou and J. Tsitsiklis. The complexity of markov decision processes. *Mathematics of Operations Research*, 12(3):441–450, 1987.

[63] C. Pecheur and R. Simmons. From livingstone to smv: Formal verification for autonomous spacecrafts. In *Proceedings of the First Goddard Workshop on Formal Approaches to Agent-Based Systems (FAABS)*, Greenbelt, MD, April 2000.

[64] B. Pell et al. A hybrid procedural/deductive executive for autonomous spacecraft. In *Proceedings of the Second International Conference on Autonomous Agents*, 1998.

[65] B. Pell et al. The Remote Agent Executive: Capabilities to support integrated robotic agents. In *Proceedings of the AAAI Spring Symposium on Integrated Robotic Architectures*, 1998.

[66] M.L. Puterman. *Markov Decision Processes.* Wiley Interscience, New York, 1994.

[67] R. Ragno. Solving optimal satisfiability problems through Clause-directed A*. M.eng. thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, Cambridge, MA, 2002.

[68] K. Rajan et al. Ground tools for the 21st century. In *Proceedings of IEEE Aerospace Conference*, Big Sky, MT, 2000.

[69] R. Rasmussen. Goal-based fault tolerance for space systems using the Mission Data System. In *Proceedings of the IEEE Aerospace Conference*, 2001.

[70] N. Rouquette and D.Dvorak. Reduced, reusable and reliable monitor software. In *Proceedings of i-SAIRAS*, 1997.

[71] N. Rouquette, P. Gluck, and R. Kanefsky. The 13th technology of Deep Space One. In *Proceedings of i-SAIRAS*, 1999.

[72] V. Saraswat, M. Rinard, and P. Panangaden. Semantic foundations of concurrent constraint programming. In *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, pages 333–352, Orlando, Florida, 1991.

[73] V. Saraswat. The category of constraint systems is cartesian-closed. In *Proceedings of the 7th IEEE Symposium on Logic in Computer Science*, 1992.

[74] V. Saraswat, R. Jagadeesan, and V. Gupta. Foundations of Timed Default Concurrent Constraint programming. In *Proceedings of the 9th IEEE Symposium on Logic in Computer Science*, 1994.

[75] M. Schoppers. Universal plans for reactive robots in unpredictable environments. In *Proceedings of IJCAI-87*, volume 2, pages 1039–1046, 1987.

[76] R. Simmons and D. Apfelbaum. A Task Description Language for robot control. In *Proceedings of the Conference on Intelligent Robotics and Systems*, Vancouver, Canada, October 1998.

[77] E. Sondik. *The Optimal Control of Partially Observvable Markov Decision Processes.* PhD thesis, Stanford University, Stanford, CA, 1971.

[78] I. Tsamardinos, N. Muscettola, and P. Morris. Fast transformation of temporal plans for efficient execution. In *Proceedings of AAAI-98*, 1998.

[79] J. Van Eepoel. Achieving real-time mode estimation through offline compilation. S.m. thesis, Massachusetts Institute of Technology, Department of Aeronautics and Astronautics, Cambridge, MA, 2002.

[80] D. Watson. Model-based autonomy in deep space missions. *IEEE Intelligent Systems*, 18(3):8–11, May/June 2003.

[81] A. Wehowsky. Safe distributed coordination of heterogeneous robots through dynamic simple temporal networks. S.m. thesis, Massachusetts Institute of Technology, Department of Aeronautics and Astronautics, Cambridge, MA, 2003.

[82] D.S. Weld. An introduction to least commitment planning. *AI Magazine*, 15(4):27–61, 1994.

[83] B.C. Williams and P. Nayak. A model-based approach to reactive self-configuring systems. In *Proceedings of AAAI-96*, volume 2, pages 971–978. AAAI Press, 1996.

[84] B.C. Williams and P. Nayak. Immobile robots: AI in the new millennium. *AI Magazine*, 17(3):16–35, 1996.

[85] B.C. Williams and P. Nayak. A reactive planner for a model-based executive. In *Proceedings of IJCAI-97*, volume 2, pages 1178–1185, 1997.

[86] B. C. Williams and V. Gupta. Unifying model-based and reactive programming within a model-based executive. In *Proceedings of DX-99*, 1999.

[87] B.C. Williams, S. Chung, and V. Gupta. Mode estimation of model-based programs: Monitoring systems with complex behavior. In *Proceedings of IJCAI-01*, 2001.

[88] B.C. Williams et al. Model-based reactive programming of cooperative vehicles for mars exploration. In *Proceedings of ISAIRAS-01*, 2001.

[89] B.C. Williams and M.D. Ingham. Model-based programming: Controlling embedded systems by reasoning about hidden state. In *Proceedings of the 8th*

International Conference on Principles and Practice of Constraint Programming (CP-02), September 2002.

[90] B.C. Williams, M. Ingham, S. Chung, and P. Elliott. Model-based programming of intelligent embedded systems and robotic space explorers. *Proceedings of the IEEE*, 91(1):212–237, January 2003.

[91] B.C. Williams and R.J. Ragno. Conflict-directed A* and its role in model-based embedded systems. *Journal of Discrete Applied Mathematics*, 2003. Forthcoming.

[92] T. Young et al. Report of the Mars Program Independent Assessment Team. Technical report, NASA, March 2000.

Appendix A

Chandra X-Ray Telescope Activation Sequence

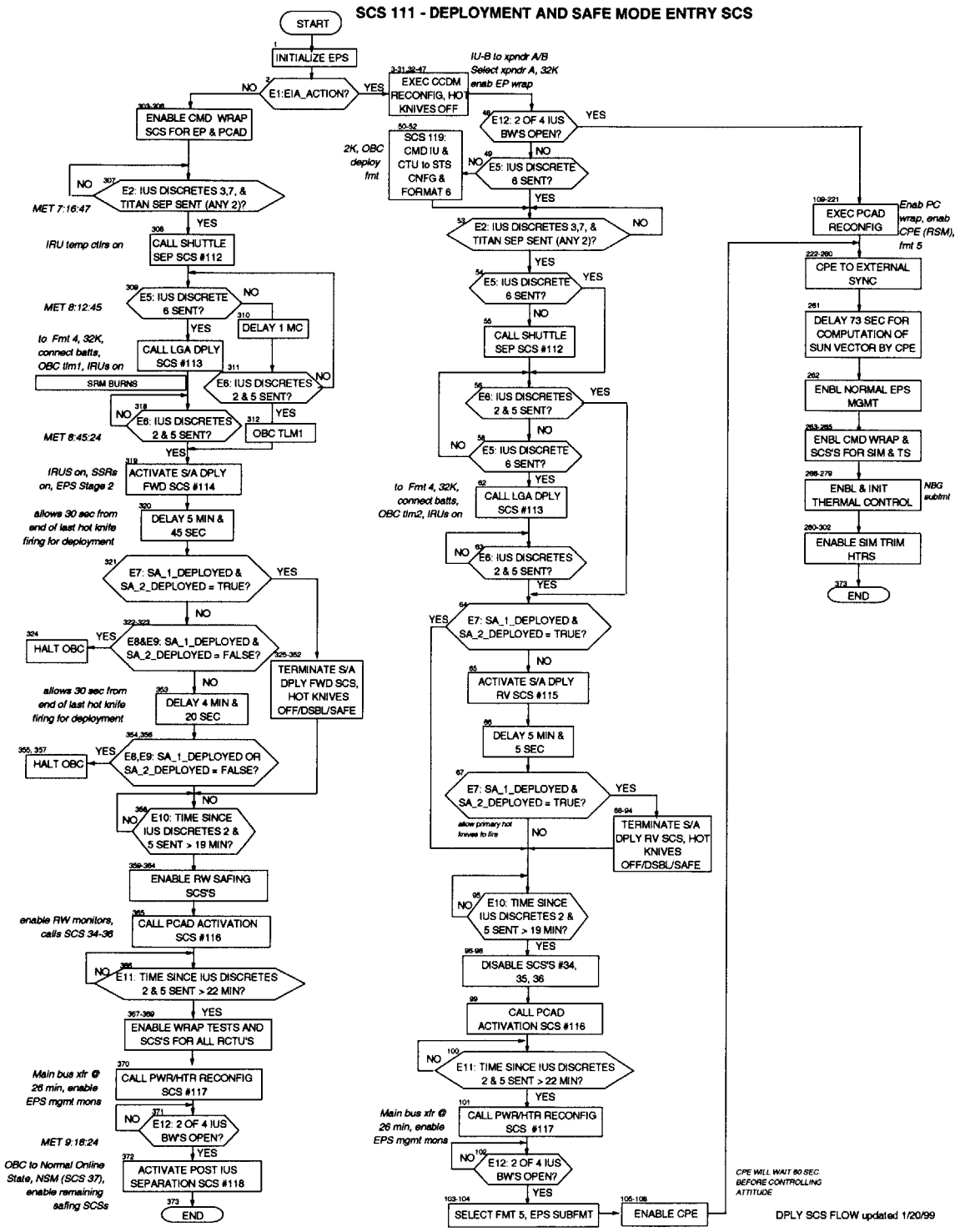

Figure A-1: Activation sequence flowchart for the Chandra X-Ray space telescope (copyright Northrop Grumman Space Technology).

Appendix B

RMPL Control Programs for Mars EDL

```
1   MarsEDL():: {
2     do {
3       EntrySequence(),
4       DescentLandingSequence()
5     } watching (landing = success)
6   }
```

Figure B-1: Main RMPL control program for the Mars EDL scenario. MarsEDL calls the EntrySequence and DescentLandingSequence subprograms.

```
1   EntrySequence():: {
2     engine = standby;
3     t1 = 0;
4     when (t1 ≥ 16200.0) donext {
5       nav = inertial;
6       t2 = 0;
7       when (t2 ≥ 240.0) donext {
8         do {
9           always (att = entry-orient),
10          when (att = entry-orient) donext (lander = separated)
11        } watching (entry = initiated)
12      }
13    }
14  }
```

Figure B-2: RMPL subprogram for the Entry Sequence.

```
1   DescentLandingSequence():: {
2     when (entry = initiated) donext {
3       lander = separated,
4       when (lander = separated) donext {
5         do {
6           always (att = vel-vector-orient)
7         } watching (Mach_trigger = triggered),
8         when (Mach_trigger = triggered) donext {
9           chute = deployed;
10          t3 = 0;
11          when (t3 ≥ 10.0) donext {
12            heatshield = jettison;
13            {
14              do {
15                always (att = vel-vector-orient)
16              } watching (alt_vel_trigger = triggered),
17              {
18                t4 = 0;
19                when (t4 ≥ 10.0) donext {
20                  legs = deployed;
21                  t5 = 0;
22                  when (t5 ≥ 1.5) donext (radar_altim = on)
23                }
24              },
25              PoweredDescentLandingSequence()
26            }
27          }
28        }
29      }
30    }
31  }
```

Figure B-3: RMPL subprogram for the Descent and Landing Sequence. DescentLandingSequence calls PoweredDescentLandingSequence.

```
1   PoweredDescentLandingSequence():: {
2     when (alt_vel_trigger = triggered) donext {
3       backshield = jettisoned;
4       t6 = 0;
5       when (t6 ≥ 0.5) donext {
6         engine = firing;
7         {
8           always (att = landing-orient),
9           do {
10            always (prop = gravity-turn)
11          } watching (alt_12m_trigger = triggered),
12          when (alt_40m_trigger = triggered) donext (radar_altim = off),
13          when (alt_12m_trigger = triggered) donext (prop = const-vel-descent),
14          whenever ((alt_12m_trigger = not-triggered) ∧ (touchdown_sensor = triggered)) donext
15            touchdown_sensor = not-triggered,
16          when ((alt_12m_trigger = triggered) ∧ (touchdown_sensor = triggered)) donext {
17            engine = off;
18            landing = success
19          }
20        }
21      }
22    }
23  }
```

Figure B-4: RMPL subprogram for the Powered Descent and Landing Sequence.

Appendix C

TCCA Plant Models for Mars EDL

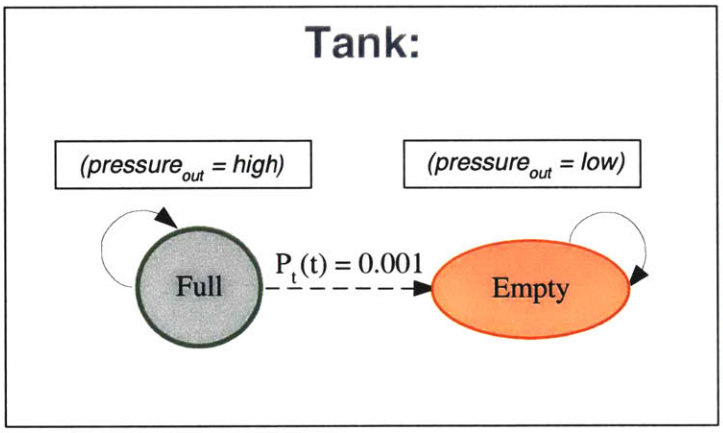

Figure C-1: Timed Constraint Automaton for the *tank* component.

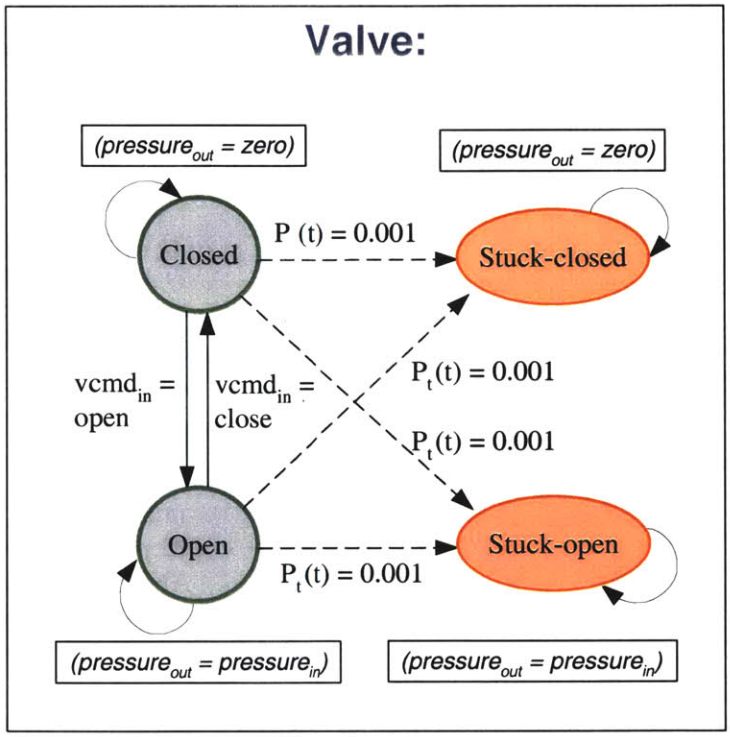

Figure C-2: Timed Constraint Automaton for the *valve* component.

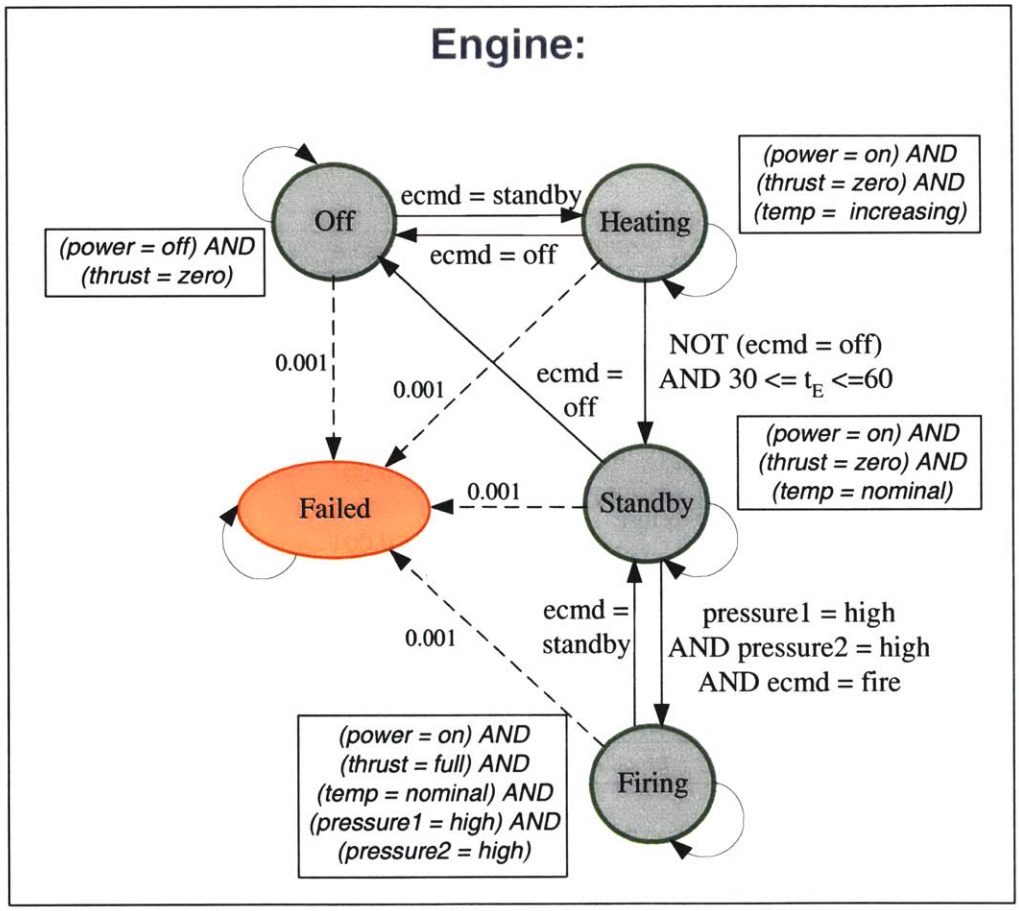

Figure C-3: Timed Constraint Automaton for the *engine* component.

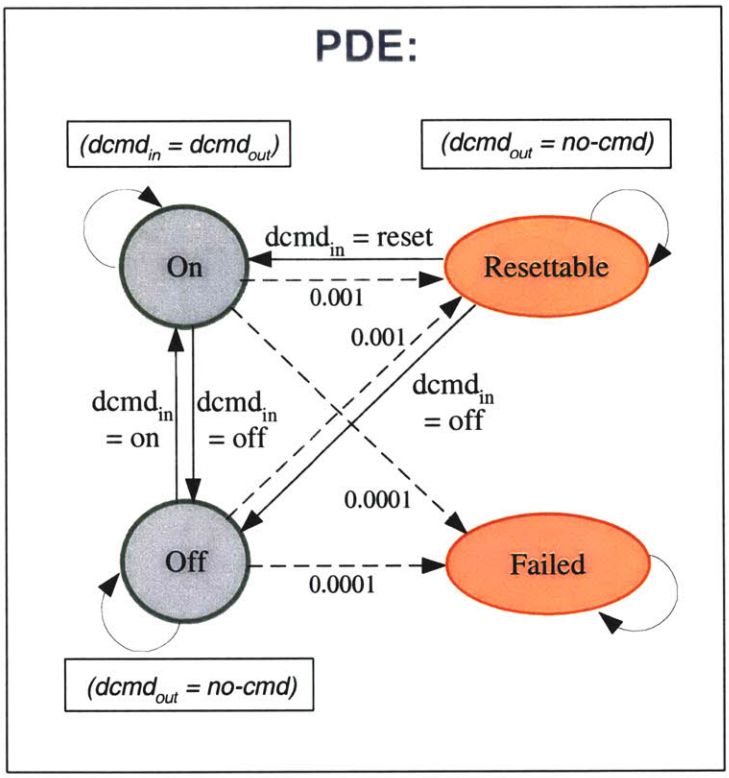

Figure C-4: Timed Constraint Automaton for the *PDE* component.

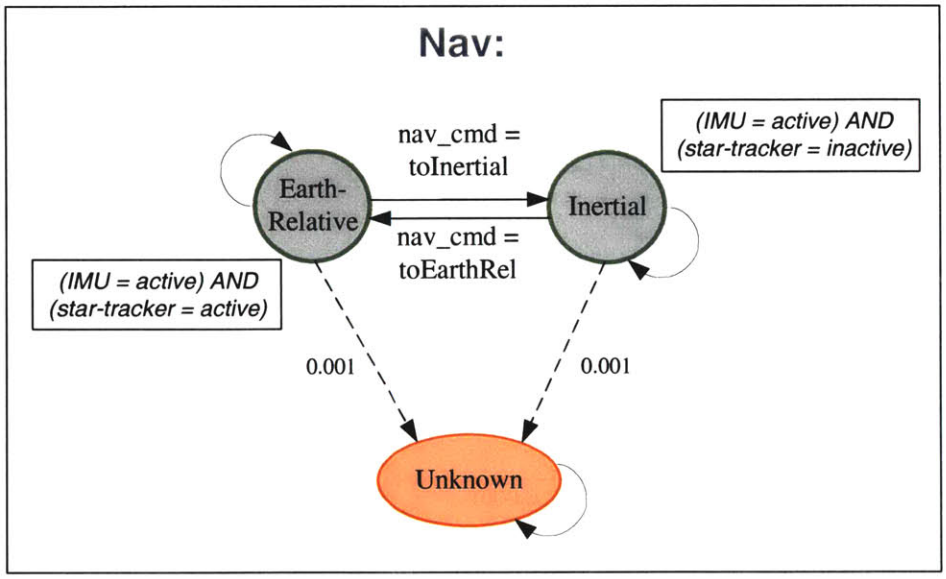

Figure C-5: Timed Constraint Automaton for the *nav* estimator mode variable.

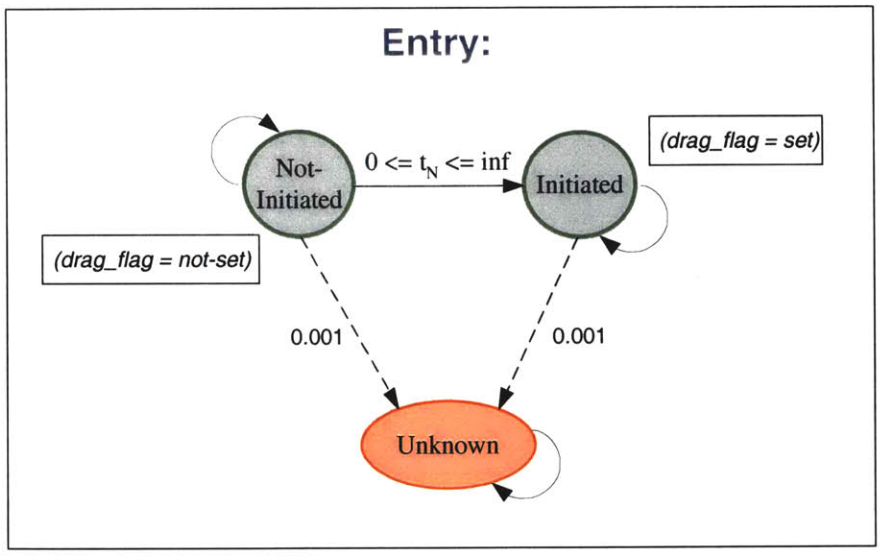

Figure C-6: Timed Constraint Automaton for the *entry* event flag.

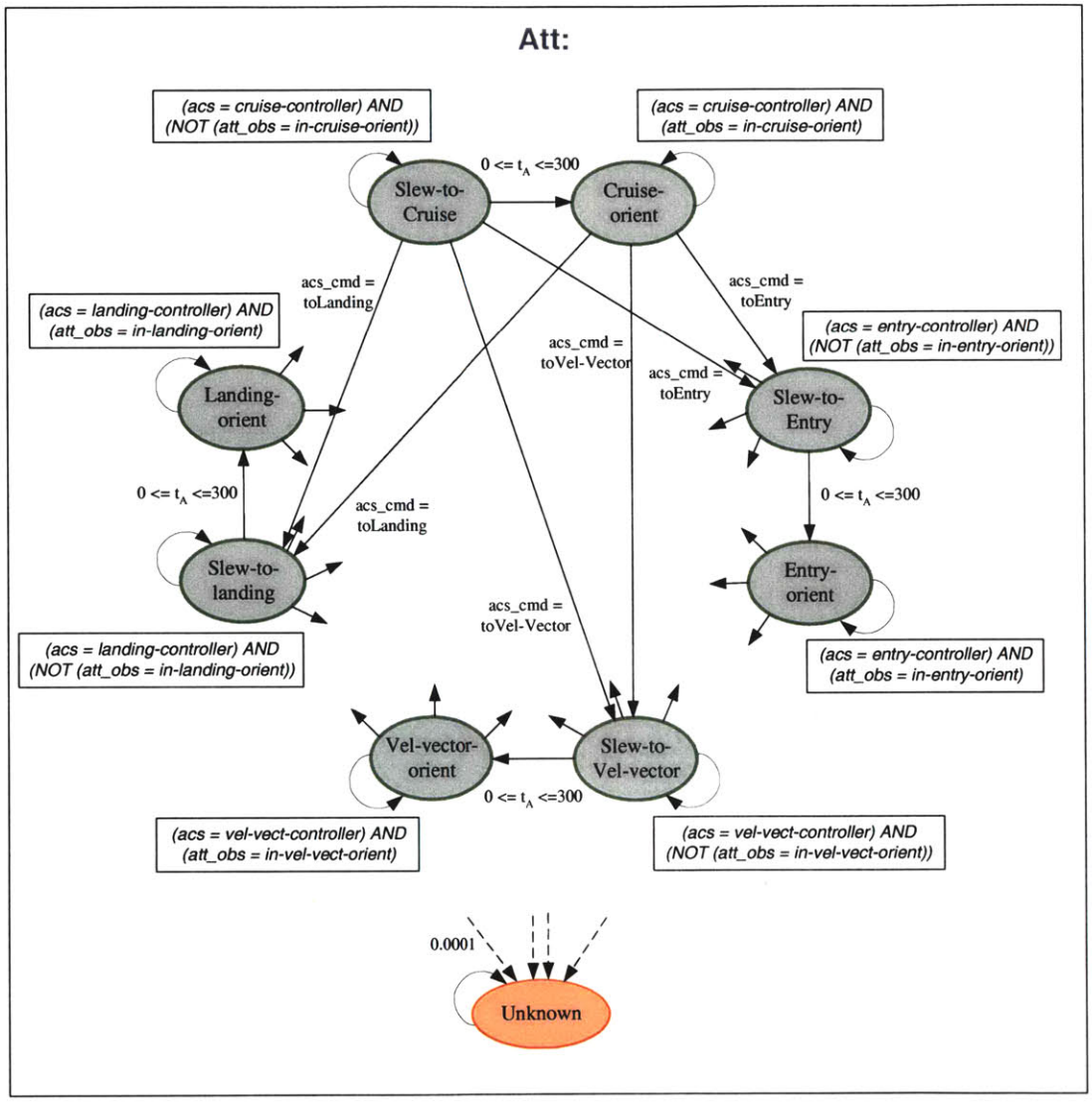

Figure C-7: Timed Constraint Automaton for the *att* state variable.

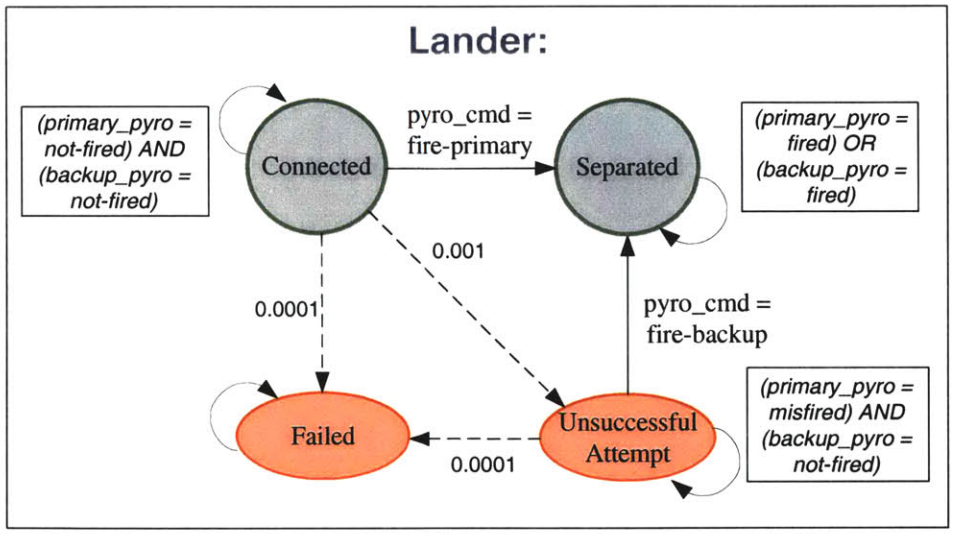

Figure C-8: Timed Constraint Automaton for the *lander* separation pyro component.

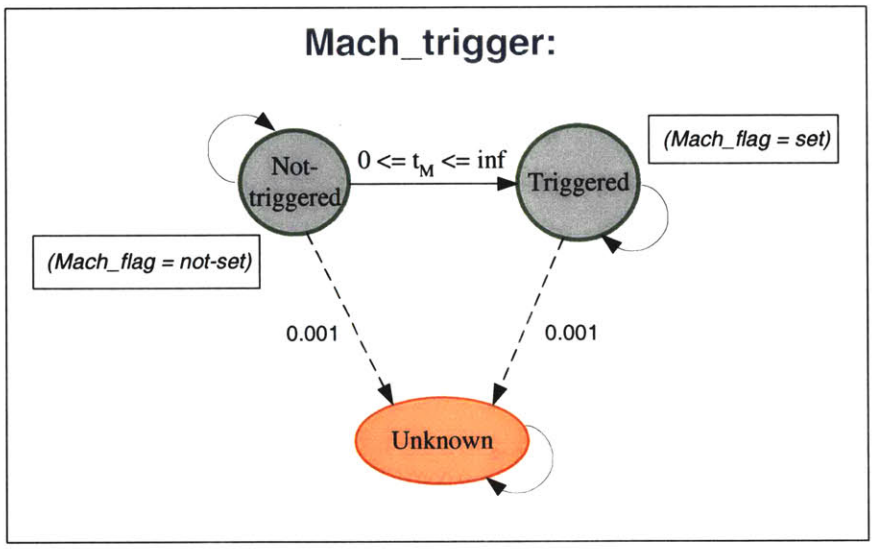

Figure C-9: Timed Constraint Automaton for the *Mach_trigger* event flag.

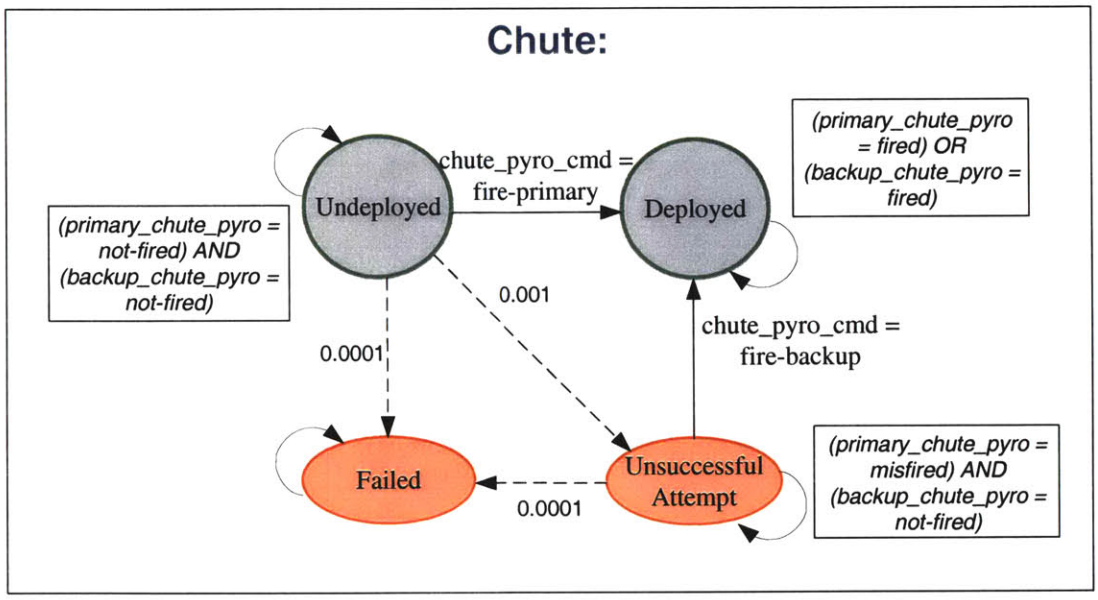

Figure C-10: Timed Constraint Automaton for the *chute* pyro component.

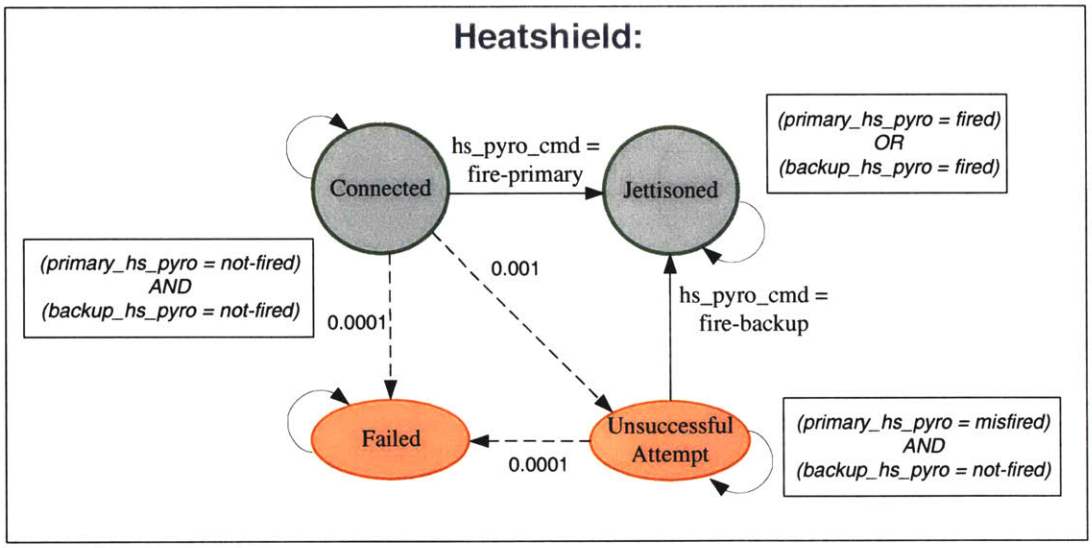

Figure C-11: Timed Constraint Automaton for the *heatshield* pyro component.

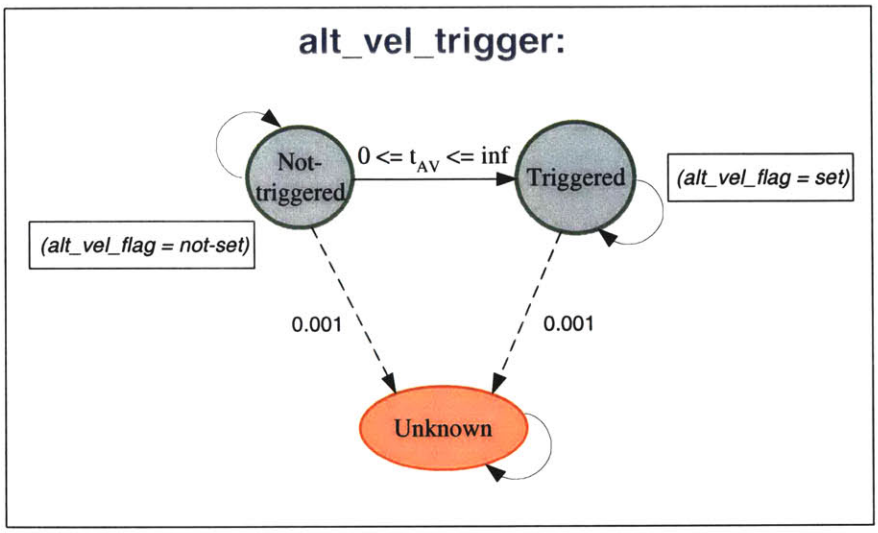

Figure C-12: Timed Constraint Automaton for the *alt_vel_trigger* event flag.

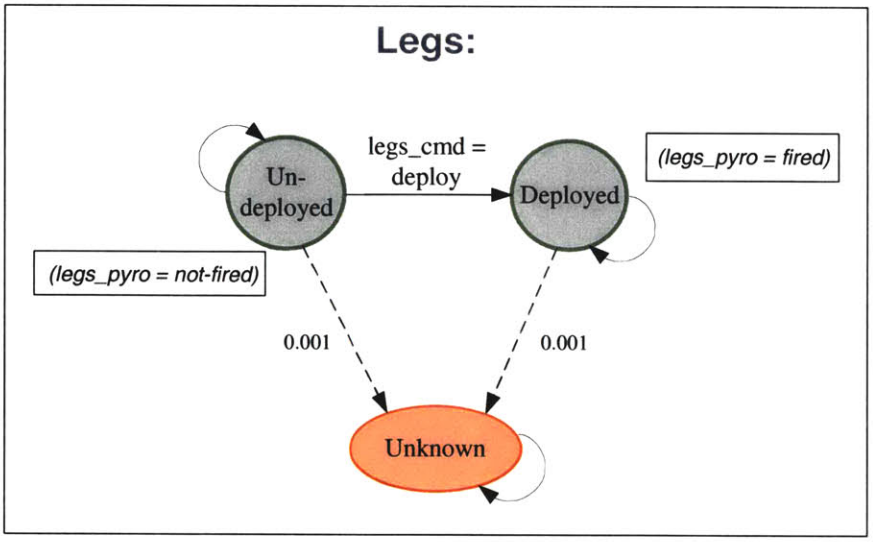

Figure C-13: Timed Constraint Automaton for the *legs* pyro component.

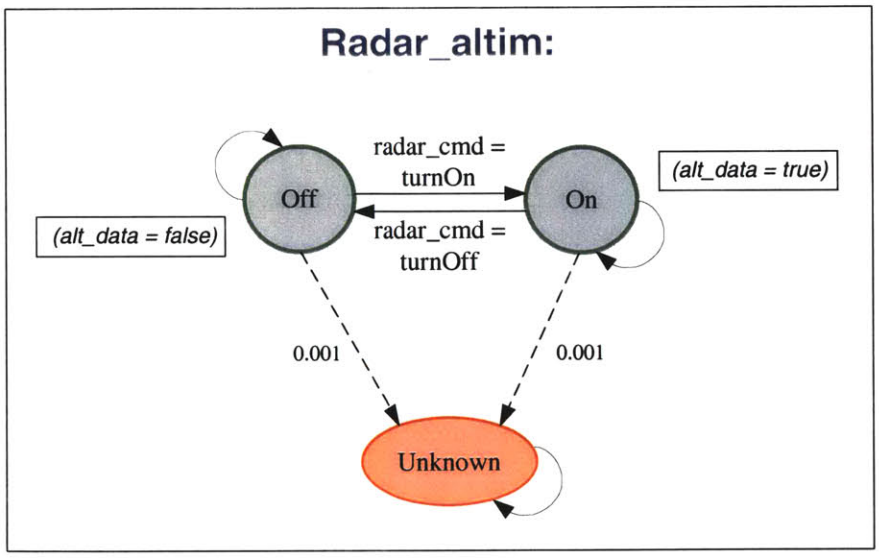

Figure C-14: Timed Constraint Automaton for the *radar_altim* sensor component.

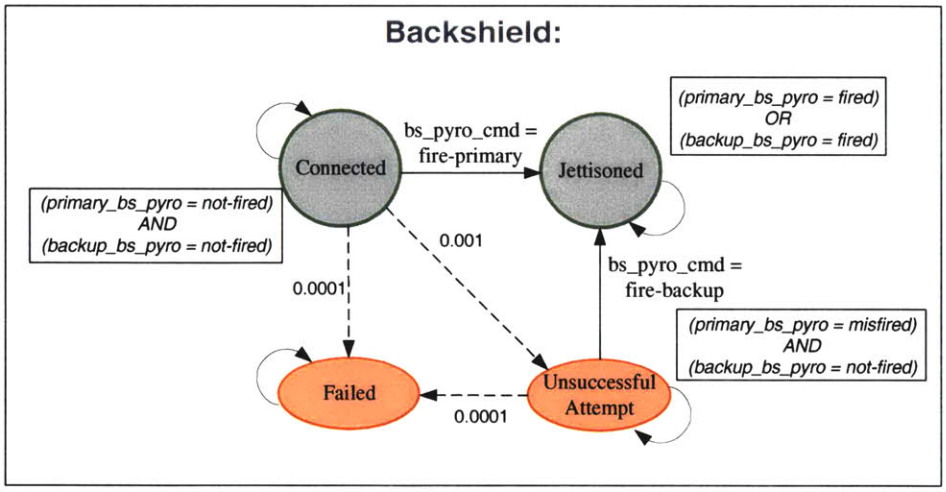

Figure C-15: Timed Constraint Automaton for the *backshield* pyro component.

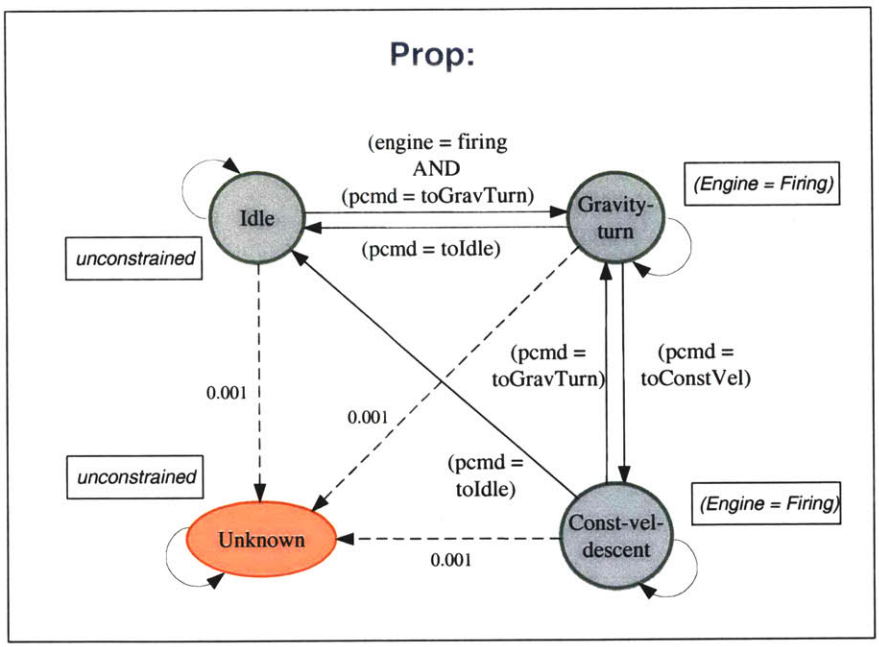

Figure C-16: Timed Constraint Automaton for the *prop* controller mode variable.

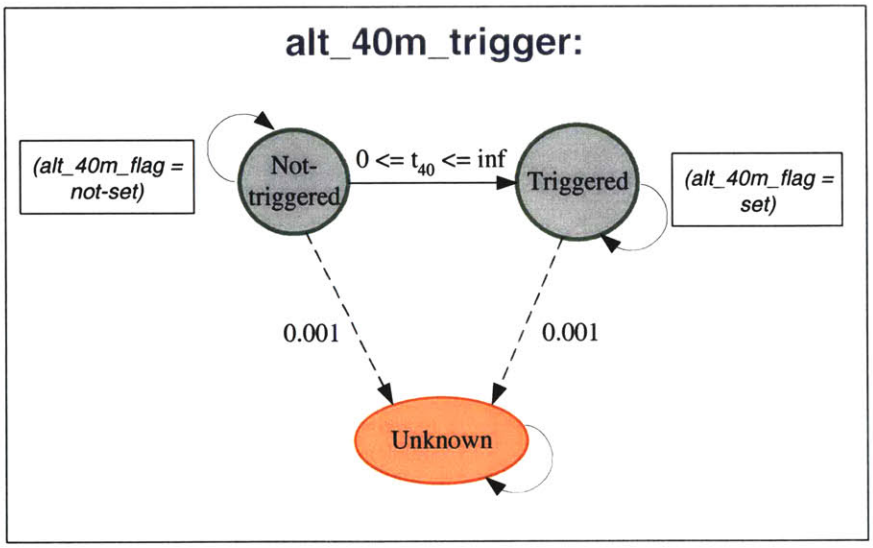

Figure C-17: Timed Constraint Automaton for the *alt_40m_trigger* event flag.

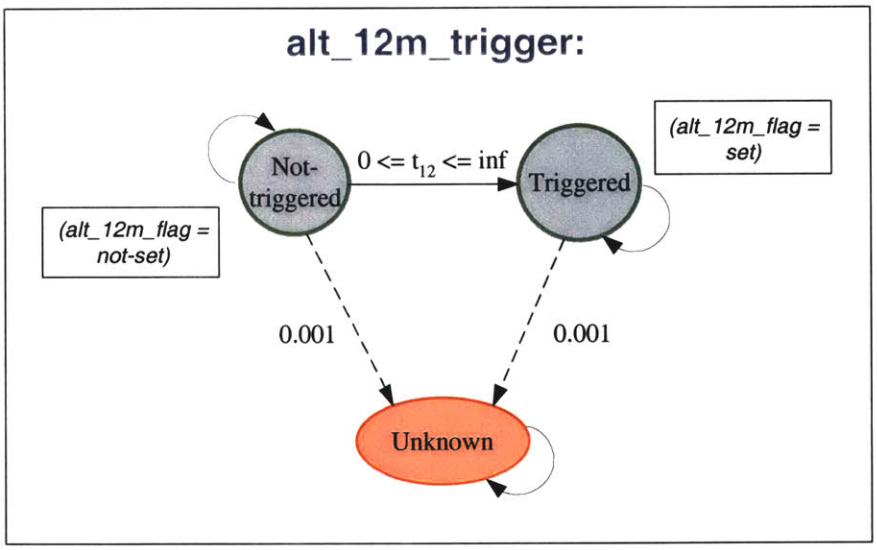

Figure C-18: Timed Constraint Automaton for the *alt_12m_trigger* event flag.

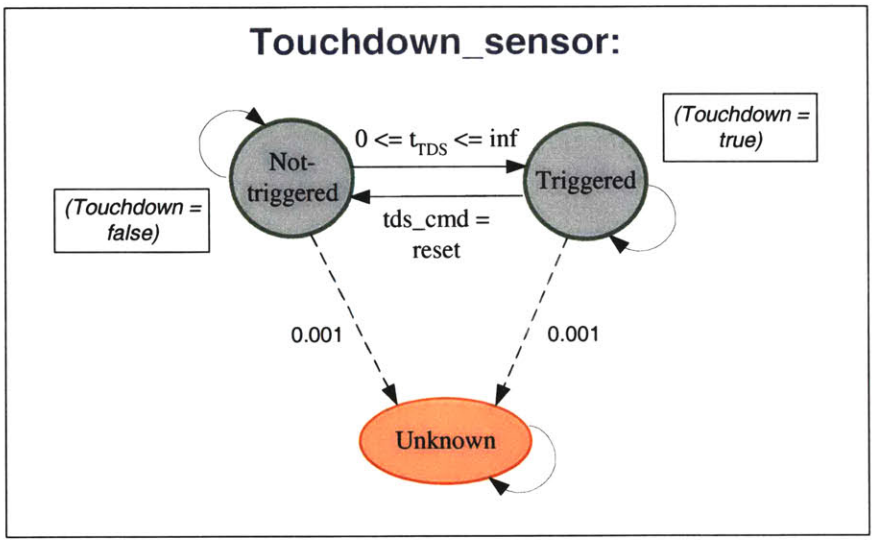

Figure C-19: Timed Constraint Automaton for the *touchdown_sensor* component.

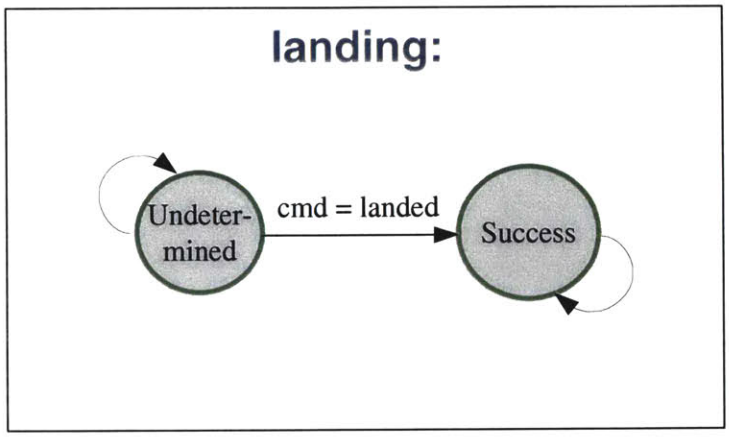

Figure C-20: Timed Constraint Automaton for the *landing* event flag.