Model-Based Programming: Controlling Embedded Systems by Reasoning About Hidden State

Brian C. Williams and Michel D. Ingham

Space Systems and Artificial Intelligence Laboratories
Massachusetts Institute of Technology
77 Massachusetts Ave., Cambridge, MA 02139
{williams, ingham}@mit.edu

Abstract. Programming complex embedded systems involves reasoning through intricate system interactions along paths between sensors, actuators and control processors. This is a time-consuming and error-prone process. Furthermore, the resulting code generally lacks modularity and robustness. *Model-based programming* addresses these limitations, allowing engineers to program by specifying high-level control strategies and by assembling common-sense models of the system hardware and software. To execute a control strategy, model-based executives reason about the models "on the fly", to track system state, diagnose faults and perform reconfigurations. This paper describes the *Reactive Model-based Programming Language (RMPL)* and its executive, called *Titan*. RMPL provides the features of synchronous reactive languages within a constraint-based modeling framework, with the added ability of being able to read and write to state variables that are hidden within the physical plant.

1 Introduction

We envision a future with large networks of highly robust and increasingly autonomous embedded systems. These visions include intelligent highways that reduce congestion, cooperative networks of air vehicles for search and rescue, and fleets of intelligent space probes that autonomously explore the far reaches of the solar system.

Many of these systems will need to perform robustly within extremely harsh and uncertain environments, or operate for years with minimal attention. To accomplish this, these embedded systems will need to radically reconfigure themselves in response to failures, and then accommodate these failures during their remaining operational lifetime. We support the rapid prototyping of these systems by creating embedded programming languages that are able to reason about how to control hardware from engineering models. This approach, which combines constraint-based and Markov modeling with the features of reactive programming, is called *model-based programming*.

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 Jupiter probe, have achieved robustness by employing sizable software development and operations teams. Efforts to make these missions highly capable at dramatically reduced costs have proven extremely challenging, producing notable losses, such as the Mars Polar Lander and Mars Climate Orbiter failures[1]. A primary contributor to these failures was the inability of the small software team to think through the large space of potential interactions between the embedded software and its underlying hardware.

Our objective is to support future programmers with embedded languages that avoid common-sense mistakes by automatically reasoning from hardware models. Our solution to this challenge has two parts. First, we have created increasingly intelligent, embedded systems that automatically diagnose and plan courses of action at reactive timescales, based on models of themselves and their environment[2–6]. This paradigm, called *model-based autonomy*, has been demonstrated in space on NASA's Deep Space One probe[7], and on several subsequent space systems[8, 9]. Second, we elevate the level at which an engineer programs through a language, called the *Reactive Model-based Programming Language (RMPL),* which enables the programmer to tap into and guide the reasoning methods of model-based autonomy. This language allows the programmer to delegate, to the language's compiler and run-time kernel, tasks involving reasoning through system interactions, such as low-level commanding, monitoring, diagnosis and repair. The model-based execution kernel for RMPL is called *Titan*.

This paper begins by describing the model-based programming paradigm in more detail (Section 2). Section 3 then goes on to demonstrate model-based programming as applied to a simple example. Section 4 introduces the RMPL language. Section 5 presents the semantics of model-based program execution. Section 6 describes Titan's control sequencer, which translates a control program written in RMPL into a sequence of state configuration goals, based on the system's estimated state trajectory. Finally, Section 7 closes with related work.

2 Model-Based Programming

Engineers like to reason about embedded systems in terms of state evolutions. However, embedded programming languages, such as Esterel[10] and Statecharts[11], interact with a physical plant by reading sensor variables and writing control variables (left, Figure 1). Constraint programming languages, such as the Timed Concurrent Constraint Language (TCC)[12], replace the traditional notion of a "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 (middle, Figure 1). In both these cases, 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 embedded programming and constraint programming, with the key difference that it interacts directly with the plant state (right, Figure 1). This is accomplished by allowing the programmer to *read or write constraints on "hidden" state variables in the plant*, i.e. states that are not directly observable or controllable. It is then the responsibility of the language's execution kernel to map between hidden states and the plant sensors and

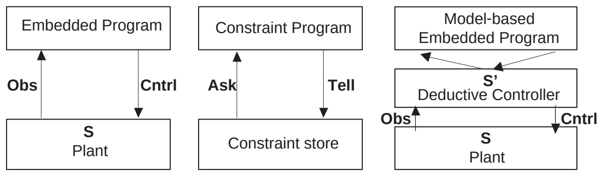

Fig. 1. Model of interaction for traditional embedded languages (left), concurrent constraint programming languages (middle) and model-based programming languages (right).

control variables. This mapping is performed automatically by employing a deductive controller that reasons from a common-sense plant model.

A model-based program is comprised of two components. The first is a *control program*, which uses standard programming constructs to codify specifications of desired system behavior. In addition, to execute the control program, the execution kernel needs a model of the system it must control. Hence the second component is a *plant model*, which includes models of the plant's nominal behavior and common failure modes. This model unifies constraints, concurrency and Markov processes.

A model-based program is executed by automatically generating a control sequence that moves the physical plant to the states specified by the control program (Figure 2). We call these specified states *configuration goals*. Program execution is performed using a *model-based executive*, such as Titan, which repeatedly generates the next configuration goal, and then generates a sequence of control actions that achieve this goal, based on knowledge of the current plant state and plant model.

The Titan model-based executive consists of two components, a *control sequencer* and a *deductive controller*. The control sequencer is responsible for generating a sequence of configuration goals, using the control program and plant state estimates. Each configuration goal specifies an abstract state for the plant to be placed in. The deductive controller is responsible for estimating the plant's most likely current state based on observations from the plant (*mode estimation*), and for issuing commands to move the plant through a sequence of states that achieve the configuration goals (*mode reconfiguration*).

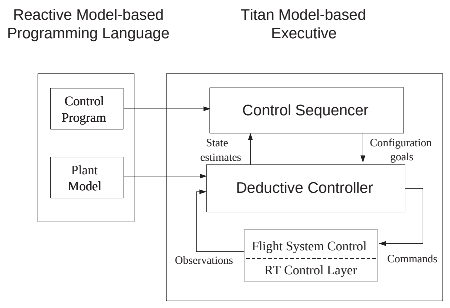

Fig. 2. Architecture for a model-based executive.

3 A Model-Based Programming Example

Model-based programming enables a programmer to focus on specifying the desired state evolutions of the system. For example, consider the task of inserting a spacecraft into orbit around a planet. Our spacecraft includes a science camera and two identical redundant engines (Engines A and B), as shown in Figure 3. An engineer thinks about this maneuver in terms of state trajectories:

> Heat up both engines (called “standby mode”). Meanwhile, turn the camera off, in order to avoid plume contamination. When both are accomplished, thrust one of the two engines, using the other as backup in case of primary engine failure.

This specification is far simpler than a control program that must turn on heaters and valve drivers, open valves and interpret sensor readings for the engine. Thinking 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 failures as they arise. This is essential for achieving high levels of robustness.

As an example, consider the model-based program for spacecraft orbital insertion. The spacecraft dual main engine system (Figure 3) consists of two propellant tanks, two main engines and redundant valves. The system offers a range of configurations for establishing propellant paths to a main engine. When the propellants combine within the engine they produce thrust. The flight computer controls the engine and camera by sending commands. Sensors include an accelerometer, to confirm engine operation, and a camera shutter position sensor, to confirm camera operation.

Control Program – The RMPL control program, shown in Figure 4, codifies the informal specification we gave above as a set of state trajectories. The specific RMPL constructs used in the program are introduced in Section 4. Recall that, to perform orbital insertion, one of the two engines must be fired. We start by concurrently placing the two engines in standby and by shutting off the camera. This is performed by lines 3-5, where the comma at the end of each line denotes parallel composition. We then fire an engine, choosing to use Engine A as the primary engine (lines 6-9) and Engine B as a backup, in the event that Engine A fails to fire correctly (lines 10-11). Engine A starts trying to fire as soon as it achieves standby and the camera is off (line 7), but aborts if at any time Engine A is found to be in a failure state (line 9). Engine B starts trying to fire only if Engine A has failed, B is in standby and the camera is off (line 10).

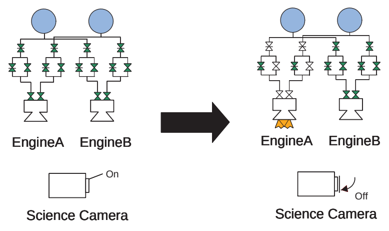

Fig. 3. Simple spacecraft for orbital insertion.

```
1  OrbitInsert  ():: {
2   do
3      { EngineA   = Standby,
4        EngineB   = Standby,
5       Camera = Off,
6        do
7            when  EngineA   = Standby AND Camera = Off
8                donext  EngineA   = Firing
9        watching  EngineA   = Failed,
10       when  EngineA   = Failed AND   EngineB   = Standby AND Camera = Off
11           donext  EngineB   = Firing}
12  watching  EngineA   = Firing OR   EngineB   = Firing
13 }
```

Fig. 4. RMPL control program for the orbital insertion scenario.

Several features of this control program reinforce our earlier points. First, the program is stated in terms of state assignments to the engines and camera, such as "EngineB = Firing". Second, these state assignments appear both as assertions and as execution conditions. For example, in lines 6-9, "EngineA = Firing" appears in an assertion (line 8), while "EngineA = Standby," "Camera = Off" and "EngineA = Failed," appear in execution conditions (lines 7 and 9). Third, none of these state assignments are directly observable or controllable, only shutter position and acceleration may be directly sensed, and only the flight computer command may be directly set. 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.

Plant Model – The plant model is used by a model-based executive to map sensed variables in the control program to queried states and asserted states to specific control sequences. The plant model is specified as a concurrent transition system, composed of probabilistic component automata[2]. Each component automaton is represented by a set of component modes, a set of constraints defining the behavior within each mode, and a set of probabilistic transitions between modes. Constraints are used to represent co-temporal interactions between state variables and inter-communication between components. Probabilistic transitions are used to model the stochastic behavior of components, such as failure and intermittency. Reward is used to assess the costs and benefits associated with particular component modes. The component automata operate concurrently and synchronously.

For example, we can model the spacecraft abstractly as a three component system (2 engines and a camera), by supplying the models depicted graphically in Figure 5. Nominally, an engine can be in one of three modes: *off*, *standby* or *firing*. The behavior within each of these modes is described by a set of constraints on plant variables, namely *thrust* and *power_in*. In Figure 5 these constraints are specified in boxes next to their respective modes. The engine also has a *failed* mode, capturing any off-nominal behavior. We entertain the possibility of a novel engine failure by specifying no constraints for the engine's behavior in the *failed* mode[13].

Models include commanded and uncommanded transitions, both of which are probabilistic. For example, the engine has uncommanded transitions from *off*, *standby* and *firing* to *failed*. These transitions have a 1% probability, indicated as arcs labeled 0.01.

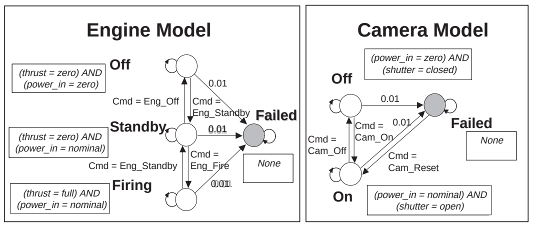
Engine Model
(thrust = zero) AND (power_in = zero)
Off
0.01
Cmd = Eng_Off
Cmd = Eng_Standby
Failed
Standby
(thrust = zero) AND (power_in = nominal)
0.01
Cmd = Eng_Standby
Cmd = Eng_Fire
None
Firing
(thrust = full) AND (power_in = nominal)
0.01
Camera Model
(power_in = zero) AND (shutter = closed)
Off
0.01
Cmd = Cam_On
0.01
Failed
Cmd = Cam_Off
None
Cmd = Cam_Reset
On
(power_in = nominal) AND (shutter = open)

Fig. 5. State transition models for a simplified spacecraft.

Transitions between nominal modes are triggered on commands, and occur with probability 99%.

Executing the Model-Based Program – When the orbital insertion control program is executed, the control sequencer starts by generating a configuration goal consisting of the conjunction of three state assignments: "EngineA = Standby", "EngineB = Standby" and "Camera = Off" (lines 3-5). To determine how to achieve this goal, the deductive controller considers the latest estimate of the state of the plant. For example, suppose the deductive controller determines from its sensor measurements and previous commands that the two engines are already in standby, but the camera is on. The deductive controller deduces from the model that it should send a command to the plant to turn the camera off. After executing this command, it uses its shutter position sensor to confirm that the camera is off. With "Camera = Off" and "EngineA = Standby", the control sequencer advances to the configuration goal of "EngineA = Firing" (line 8). The deductive controller identifies an appropriate setting of valve states that achieves this behavior, then it sends out the appropriate commands.

In the process of achieving goal "EngineA = Firing", assume that a failure occurs: an inlet valve to Engine A suddenly sticks closed. Given various sensor measurements (e.g. flow and pressure measurements throughout the propulsion subsystem), the deductive controller identifies the stuck valve as the most likely source of failure. It then tries to execute an alternative control sequence for achieving the configuration goal, for example, by repairing the valve. Presume that the valve is not repairable; Titan diagnoses that "EngineA = Failed". The control program specifies a configuration goal of "EngineB = Firing" as a backup (lines 10-11), which is issued by the control sequencer to the deductive controller.

4 The Reactive Model-Based Programming Language

RMPL is an object-oriented language, like Java. In general, RMPL constructs are conditioned on the current state of the physical plant, and they act on the plant state in the next time instant. State assertions are specified as constraints on plant state variables that should be made true. RMPL's model of interaction is in contrast to Esterel and TCC, which both interact with the program memory, sensors and control variables, but not with the plant state. Esterel interacts by emitting and detecting signals, while TCC interacts by *telling* and *asking* constraints on program variables. In contrast, RMPL constructs *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 checks are *ask* operations.

RMPL currently uses propositional state logic for its constraint system. In propositional state logic each proposition is an assignment, $x = v$, where variable x ranges over a finite domain $\mathbb{D}(x)$. 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 plant model and the most likely current state of the physical plant.

We introduce RMPL by first highlighting its desired features. We then present the RMPL constructs used to encode control programs. These constructs are fully orthogonal, that is, they may be nested and combined arbitrarily. RMPL constructs are closely related to the TCC programming language constructs[12].

The orbital insertion example highlights five design features of RMPL. First, the program exploits full concurrency, by specifying parallel threads of execution; for example, the camera is turned off and the engines are set to standby in parallel (lines 3-5). Second, it involves conditional execution; for example, the control program must check for two conditions, prior to firing Engine A: that the engine to be fired is in standby mode, and that the camera is turned off (line 7). Third, it involves iteration; for example, line 7 says to iteratively test until Engine A is in standby and the Camera is off, and then to proceed. Fourth, the program involves preemption; for example, if the primary engine fails, the act of firing it should be preempted, in favor of firing the backup engine (lines 6-9). These four features are common to most synchronous reactive programming languages. As highlighted previously, the fifth and defining feature of RMPL is the ability to reference hidden states of the physical plant within assertions and condition checks, such as "**when** EngineA = Standby $\wedge$ Camera = Off **donext** EngineA = Firing" (lines 7-8).

The RMPL constructs are defined as follows. We use lower case letters, like c, to denote constraints on variables of the physical plant, and upper case letters, like A and B, to denote well-formed RMPL expressions:

– g [**maintaining** m]. Asserts that the plant should progress towards a state that achieves g, while maintaining m throughout. If m does not hold at any point, then assertion of g terminates immediately. "maintaining m" is optional (defaults to **True**).
– **if** c **thennext** A_{then}[**elsenext** A_{else}]. Starts executing A_{then} in the next instant, if the most likely current plant state entails c. The optional expression A_{else} is executed starting in the next instant if c is *not* entailed by the most likely current state.
– **unless** c **thennext** A. Starts executing A in the next instant if the current theory does *not* entail c.
– A, B. Concurrently executes A and B, starting in the current instant.
– **always** A. Starts a new copy of A at each instant of time, for all time.
– A; B. Performs A until A is finished, then starts B.
– **when** c **donext** A. Waits until c is entailed by the most likely plant state, then starts A in the next instant.

- **whenever** c **donext** A. For every instant in which c holds for the most likely state, it starts a copy of A in the next instant.
- **do** A **watching** c. Executes A, but if c becomes entailed by the most likely plant state at any instant, it terminates execution of A.

The above-mentioned RMPL constructs are used to encode control programs. This subset is sufficient to implement most of the control constructs of the Esterel language[4]. 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 [14].

5 Model-Based Program Execution Semantics

We define the execution of a model-based program in terms of legal state evolutions of a physical plant $\mathbb{P}$.

Plant Model – A plant $\mathbb{P}$ is modeled as a *partially observable Markov decision process* (POMDP) $M = \langle \Pi, \Sigma, \mathbb{T}, P_\Theta, P_\mathbb{T}, P_O, R\rangle$. Π is a set of *variables*, each ranging over a finite domain. Π is partitioned into *state variables* Π_s, *control variables* Π_c, *observable variables* Π_o, and *dependent variables* Π_d. A *full assignment* σ is defined as a set consisting of an assignment to each variable in Π. Σ is the set of all *feasible* full assignments over Π. A *state* s is defined as an assignment to each variable in Π_s. The set Σ_s, the projection of Σ on variables in Π_s, is the set of all feasible states.

$\mathbb{T}$ is a finite set of *transitions*. Each transition $\tau \in \mathbb{T}$ is a function $\tau : \Sigma \rightarrow \Sigma_s$, *i.e.* $\tau(\sigma_i)$ is the state obtained by applying transition τ to any feasible full assignment σ_i. The transition $\tau_n \in \mathbb{T}$ models the system's nominal behavior, while all other transitions model failures. The probability of transition τ, given full assignment σ_i, is $P_\tau(\sigma_i)$, for $P_\tau \in P_\mathbb{T}$. $P_\Theta(s_0)$ is the probability that the plant has initial state s_0. The reward for being in state s_i is $R(s_i)$, and the probability of observing o_j in state s_i is $P_O(s_i, o_j)$.

A *plant trajectory* is a (finite or infinite) sequence of feasible states $S : s_0, s_1, \ldots$ such that for each s_i there is a feasible assignment $\sigma_i \in \Sigma$ which agrees with s_i on assignments to variables in Π_s and $s_{i+1} = \tau(\sigma_i)$ for some $\tau \in \mathbb{T}$. A trajectory that involves only the nominal transition τ_n is called a *nominal trajectory*. A *simple* trajectory does not repeat any state.

Model-Based Program Execution – A model-based program for plant $\mathbb{P}$ consists of a model M, described above, and a control program CP, described as a deterministic automaton $CP = \langle \Sigma_{cp}, \theta_{cp}, \tau_{cp}, g_{cp}, \Sigma_s\rangle$. Σ_{cp} is the set of *program locations*, where $\theta_{cp} \in \Sigma_{cp}$ is the program's initial location. Transitions τ_{cp} between locations are conditioned on plant states Σ_s of M, *i.e.* τ_{cp} is a function $\tau_{cp} : \Sigma_{cp} \times \Sigma_s \rightarrow \Sigma_{cp}$. Each location $l_i \in \Sigma_{cp}$ has a corresponding *configuration goal* $g_{cp}(l_i) \subset \Sigma_s$, which is the set of legal plant target states associated with location l_i.

A *legal execution* of a model-based program is a trajectory of feasible plant states, $S : s_0, s_1, \ldots$ of M, and locations $L : l_0, l_1, \ldots$ of CP such that: (a) s_0 is a valid initial plant state, that is, $P_\Theta(s_0) > 0$; (b) for each s_i there is a $\sigma_i \in \Sigma$ of M that agrees with s_i and o_i on the corresponding subsets of variables; (c) l_0 is the initial program location θ_{cp}; (d) $\langle l_i, l_{i+1}\rangle$ represents a legal control program transition, *i.e.* $l_{i+1} = \tau_{cp}(l_i, s_i)$; and (e) if plant state s_{i+1} is the result of a nominal transition from σ_i, *i.e.* $s_{i+1} = \tau_n(\sigma_i)$, then either s_{i+1} is the least-cost state in $g_{cp}(l_i)$, or $\langle s_i, s_{i+1}\rangle$ is the prefix of a simple nominal trajectory that ends in a least-cost state $s_j \in g_{cp}(l_i)$.

Model-Based Executive – A model-based program is executed by a *model-based executive*. We define a model-based executive as a high-level *control sequencer* coupled to a low-level *deductive controller*.

A control sequencer takes as inputs a control program CP and a sequence S : $s^{(0)}, s^{(1)}, \ldots$ of plant state estimates. It generates a sequence $\gamma : g^{(0)}, g^{(1)}, \ldots$ of configuration goals. A complete definition of the control sequencer is given in Section 6.

A deductive controller takes as inputs the plant model M, a sequence of configuration goals $\gamma : g^{(0)}, g^{(1)}, \ldots$, and a sequence of observations $O : o^{(0)}, o^{(1)}, \ldots$. It generates a sequence of most likely plant state estimates $S : s^{(0)}, s^{(1)}, \ldots$ and a sequence of control actions $\mu : \mu^{(0)}, \mu^{(1)}, \ldots$.

The sequence of state estimates is generated by a process called *mode estimation* (ME). ME incrementally tracks the set of state trajectories that are consistent with the plant model, the sequence of observations and the control actions. The ME process is framed as an instance of POMDP belief state update, which computes the probability associated with being in state s_i at time $t+1$ according to the following equations:

$$p^{(\bullet t+1)}[s_i] = \sum_{j=1}^{n} p^{(t\bullet)}[s_j] P_{\mathbb{T}}(\sigma_j \mapsto s_i)$$

$$p^{(t+1\bullet)}[s_i] = p^{(\bullet t+1)}[s_i] \frac{P_O(s_i, o_k)}{\sum_{j=1}^{n} p^{(\bullet t+1)}[s_j] P_O(s_j, o_k)}$$

where $P_{\mathbb{T}}(\sigma_j \mapsto s_i)$ is defined as the probability that M transitions from σ_j to state s_i, and the probability $p^{(\bullet t+1)}[s_i]$ is conditioned on all observations up to $o^{(t)}$, while $p^{(t+1\bullet)}[s_i]$ is also conditioned on the latest observation $o^{(t+1)}$. The tracked state with the highest belief state probability is selected as the most likely state $s^{(t)}$.

The sequence of control actions is generated by a process called *mode reconfiguration* (MR). MR takes as inputs a configuration goal and the most likely current state from ME, and it returns a series of commands that progress the plant towards a least-cost state that achieves the configuration goal. MR accomplishes this through two capabilities, the *goal interpreter* (GI) and *reactive planner* (RP). GI uses the plant model and the most likely current state to determine a reachable target state that achieves the configuration goal, while minimizing cost (or maximizing reward) $R(s)$. RP takes a target state and a current mode estimate, and generates a command sequence that moves the plant to this target. RP generates and executes this sequence one command at a time, using ME to confirm the effects of each command.

Since the size of the set of possible current states is exponential in the number of components, computational resource limitations only allow a small fraction of the state space to be explored in real time. ME tracks the most likely states using the OpSat optimal constraint satisfaction engine[15]. GI also uses OpSat to search for a minimum-cost target state. RP, also called *Burton*[3], is a sound, complete planner that generates a control action of a valid plan in average case constant time.

The deductive controller has been described extensively in [2, 3], in the remainder of this paper we focus on the technical details of the control sequencer.

6 Control Sequencer

The RMPL control program is executed by Titan's control sequencer. Executing a control program involves compiling it to a variant of hierarchical automata, called *hierarchical constraint automata (HCA)*, and then executing the automata in coordination with the deductive controller. In this section, we define HCA, their compilation and execution.

6.1 Hierarchical Constraint Automata

To efficiently execute RMPL programs, we translate each of the constructs introduced in Section 4 into an HCA. In the following we call the "states" of an HCA *locations*, to avoid confusion with the physical plant state. An HCA has five key attributes. First, it composes sets of concurrently operating automata. Second, each location is labeled with a constraint, called a *goal constraint*, which the physical plant must immediately begin moving towards, whenever the automaton marks that location. Third, each location is also labeled with a constraint, called a *maintenance constraint*, which must hold for that location to remain active. Fourth, 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 more complex concurrent and sequential behaviors. Finally, each transition may have multiple target locations, allowing an automaton to have several locations marked simultaneously. This enables a compact representation for iterative behaviors, like RMPL's **always** construct.

Hierarchical encodings form the basis for embedded reactive languages like Esterel[10] and State Charts[11]. A distinctive feature of an HCA is its use of constraints on plant state, in the form of goal and maintenance constraints. We elaborate on this point once we introduce HCA.

A *hierarchical, constraint automaton (HCA)* is a tuple $\langle \Sigma, \Theta, \Pi, \mathbb{G}, \mathbb{M}, \mathbb{T} \rangle$, where:

- Σ is a set of *locations*, partitioned into *primitive locations* Σ_p and *composite locations* Σ_c. Each composite location denotes a hierarchical constraint automaton.
- $\Theta \subseteq \Sigma$ is the set of *start locations* (also called the *initial marking*).
- Π is the set of plant state variables, with each $x_i \in \Pi$ ranging over a finite domain $\mathbb{D}[x_i]$. $\mathbb{C}[\Pi]$ denotes the set of all finite domain constraints over Π.
- $\mathbb{G} : \Sigma_p \rightarrow \mathbb{C}[\Pi]$, associates with each location $\sigma_i^p \in \Sigma_p$ a finite domain constraint $\mathbb{G}(\sigma_i^p)$ that the plant progresses towards whenever σ_i^p is marked. $\mathbb{G}(\sigma_i^p)$ is called the *goal constraint* of σ_i^p. Goal constraints $\mathbb{G}(\sigma_i^p)$ may be thought of as "set points", representing a set of states that the plant must evolve towards when σ_i^p is marked.
- $\mathbb{M} : \Sigma \rightarrow \mathbb{C}[\Pi]$, associates with each location $\sigma_i \in \Sigma$ a finite domain constraint $\mathbb{M}(\sigma_i)$ that must hold at the current instant for σ_i to be marked. $\mathbb{M}(\sigma_i)$ is called the *maintenance constraint* of σ_i. Maintenance constraints $\mathbb{M}(\sigma_i)$ 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_i.
- $\mathbb{T} : \Sigma \times \mathbb{C}[\Pi] \rightarrow 2^{\Sigma}$ associates with each location $\sigma_i \in \Sigma$ a transition function $\mathbb{T}(\sigma_i)$. Each $\mathbb{T}(\sigma_i) : \mathbb{C}[\Pi] \rightarrow 2^{\Sigma}$, specifies a *set* of locations to be marked at time $t + 1$, given appropriate assignments to Π at time t.

At any instant t, the "state" of an HCA is the set of marked locations $m^{(t)} \subseteq \Sigma$, called a *marking*. $\mathfrak{M}$ denotes the set of possible markings, where $\mathfrak{M} \subseteq 2^{\Sigma}$.

In the graphical representation of HCA, primitive locations are represented as circles, while composite locations are represented as rectangles. Goal and maintenance constraints are written within the corresponding locations, with maintenance constraints preceded by the keyword "maintain". Maintenance constraints can be of the form $\models c$ or $\not\models c$, for some $c \in \mathbb{C}[\Pi]$. For convenience, in our diagrams we use c to denote the constraint $\models c$, and $\overline{c}$ to denote the constraint $\not\models c$. 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 entailed by the conjunction of the plant model and the most likely estimated state of the plant. For each location σ, we represent the transition function $\mathbb{T}(\sigma)$ as a set of transition pairs (l_i, σ_i), where $\sigma_i \in \Sigma$, and l_i is a set of labels (also known as *guard conditions*) of the form $\models c$ (denoted c) or $\not\models c$ (denoted $\overline{c}$), for some $c \in \mathbb{C}[\Pi]$. This corresponds to the traditional representation of transitions as labeled arcs in a graph, where σ and σ_i are the source and target of an arc with label l_i. Again, if no label is indicated, it is implicitly **True**.

Our HCA encoding has four properties that distinguish it from the hierarchical automata employed by the above-mentioned reactive embedded languages[10, 11]. First, multiple transitions may be simultaneously traversed. This permits an exceptionally compact encoding of the state of the automaton as a set of markings. Second, transitions are conditioned on what can be deduced from the estimated 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 are enabled based on lack of information. 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.

Each RMPL construct maps to an HCA, as shown in Figure 6. For example, the RMPL code for orbital insertion (Figure 4) compiles to the HCA shown in Figure 7.

6.2 Executing HCA

To execute an HCA A, the control sequencer starts with an estimate of the current state of the plant, $s^{(0)}$. It initializes A using $m_F(\Theta(A))$, a function that marks the start locations of A and all subautomata of these start locations. It then repeatedly steps automaton A using the function $Step_{HCA}$, which maps the current state estimate and marking to a next marking and configuration goal. m_F and $Step_{HCA}$ are defined below.

Given a set of automata m to be initialized, $m_F(m)$ creates a *full marking*, by recursively marking the start locations of m and all their starting subautomata:
$m_F(m) = m \cup \bigcup \{m_F(\Theta(\sigma)) \mid \sigma \in m, \sigma \text{ is composite}\}$.

$Step_{HCA}$ transitions an automaton A from the current full marking to the next full marking and generates a new configuration goal, based on the current state estimate. That is, $Step_{HCA}(A, m^{(t)}, s^{(t)}) \rightarrow \langle g^{(t)}, m^{(t+1)}, s^{(t+1)} \rangle$ is defined by the following algorithm:

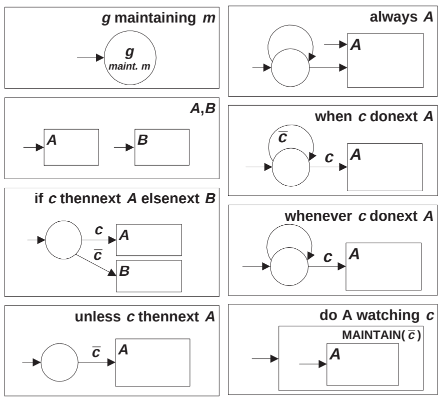

Fig. 6. Corresponding HCA for various RMPL constructs.

1. **Check maintenance constraints for marked composites.** Unmark all subautomata of any marked composite location in $m^{(t)}$ whose maintenance constraint is not entailed by $s^{(t)}$.
2. **Setup goal.** Output, as the configuration goal $g^{(t)}$, the conjunction of goal constraints from currently marked primitive locations in $m^{(t)}$ whose maintenance constraints are entailed by $s^{(t)}$.
3. **Take action.** Request that mode reconfiguration issue a command that progresses the plant towards a state s that achieves the configuration goal $g^{(t)}$.
4. **Read next state estimate.** Once the command has been issued, obtain from mode estimation the plant's new most likely state $s^{(t+1)}$.
5. **Await incomplete goals.** If the goal constraint of a primitive location marked in $m^{(t)}$ is not entailed by $s^{(t+1)}$, and its maintenance constraint was not violated by $s^{(t)}$, then include that location as marked in $m^{(t+1)}$.

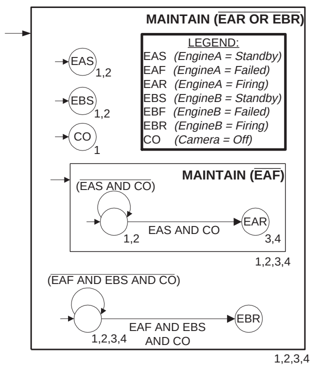

Fig. 7. HCA model for the orbital insertion scenario.

6. **Identify enabled transitions.** A transition from a marked primitive location σ_i^p in $m^{(t)}$ is enabled if both of the following conditions hold true:
   (a) σ_i^p's goal constraint is satisfied by $s^{(t+1)}$, or its maintenance constraint was violated by $s^{(t)}$;
   (b) the transition's guard condition is satisfied by $s^{(t+1)}$.

   A transition from a marked composite location σ_i^c in $m^{(t)}$ is enabled if both of the following conditions hold true:
   (a) none of σ_i^c's subautomata are marked in $m^{(t+1)}$ and none of σ_i^c's subautomata have enabled outgoing transitions;
   (b) the transition's guard condition is satisfied by $s^{(t+1)}$.
7. **Take transitions.** Mark and initialize in $m^{(t+1)}$ the target of each enabled transition. Re-mark in $m^{(t+1)}$ all composite locations with subautomata that are marked in $m^{(t+1)}$.

A's execution completes at time τ if $m^{(\tau)}$ is the empty marking, and there is no $t < \tau$ such that $m^{(t)}$ is the empty marking.

6.3 Example: Executing the Orbital Insertion Control Program

The control sequencer interacts tightly with the mode estimation and mode reconfiguration capabilities of the deductive controller, which we demonstrate with a failure-free execution trace for the orbital insertion scenario. Markings for each execution cycle are represented in Figure 7 by the numerical labels associated with each location.

Initial State – Initially, all start locations are marked (locations labeled "1", Figure 7). We assume mode estimation provides initial plant state estimates *EngineA=Off, EngineB=Off, Camera=On*.

Execution will continue as long as the maintenance constraint on the outermost composite location, $\not\models$ *(EngineA=Firing OR EngineB=Firing)*, remains true. It terminates as soon as *(EngineA=Firing OR EngineB=Firing)* is entailed. Similarly, execution of the inner composite location terminates if ever *(EngineA=Failed)* is entailed.

First Step – Since none of the maintenance constraints are violated for the initial state estimate, all start locations remain marked. The goal constraints asserted by the start locations consist of *EngineA=Standby*, *EngineB=Standby* and *Camera=Off*. These state assignments are conjoined into a configuration goal, and passed to mode reconfiguration. Mode reconfiguration issues the first command in a sequence that achieves the configuration goal. In this example, mode estimation confirms that *Camera=Off* is achieved after a one-step operation.

The locations asserting *(EngineA=Standby)* and *(EngineB=Standby)* remain marked in the next execution step, because these two configuration goals have not yet been achieved. Since *Camera=Off* has been achieved and there are no specified transitions from the primitive location that asserted this state goal, this thread of execution terminates. The other two marked primitives correspond to "**when** ... **donext** ..." expressions in the RMPL control program, they both remain marked in the next execution step, since the only enabled transitions from these locations are self-transitions. The next execution step's marking includes locations labeled with a "2" in Figure 7.

Second Step – The maintenance constraints are confirmed to hold for the current state estimate. Next, the goal constraints of marked locations are collected, *EngineA=Standby* $\wedge$ *EngineB=Standby*, and passed to mode reconfiguration. Mode reconfiguration issues the first command towards achieving this goal.

Assume that a single command is required to set both engines to standby, and that this action is successfully performed. Consequently, mode estimation indicates that the new state estimate includes *(EngineA=Standby)* and *(EngineB=Standby)*. This results in termination of the two execution threads corresponding to these goal constraint assertions. In addition, the transition labeled with condition *(EngineA=Standby AND Camera=Off)* is enabled, and hence traversed. Thus, after taking the enabled transitions, only two of the primitive locations remain marked, as shown by labels "3" in Figure 7.

Remaining Steps – The deductive controller issues and monitors a command sequence, given goal *(EngineA=Firing)*, until a flow of fuel and oxidizer are established, and mode estimation confirms that the engine is indeed firing. Since this violates the outer maintenance condition, the entire block of Figure 7 is exited.

7 Discussion

Titan is implemented in C++. The performance of its deductive controller is documented in [3, 15]. Titan is being demonstrated on the MIT SPHERES mission, and mission scenarios for NASA's ST-7 and MESSENGER missions.

Turning to related work, the model-based programming paradigm synthesizes ideas underlying synchronous programming, concurrent constraint programming, traditional robotic execution and POMDPs. Synchronous programming languages [10, 11] were developed for writing control code for reactive systems. Synchronous programming languages exhibit logical concurrency, orthogonal preemption, multiform time and determinacy, which Berry has convincingly argued are necessary characteristics for reactive programming. RMPL is a synchronous language, and satisfies all these characteristics.

Model-based programming and concurrent constraint programming share common underlying principles, including the notion of computation as deduction over systems of partial information[12, 16]. RMPL extends 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 Markov modeling. This provides a rich approach to managing discrete processes, uncertainty, failure and repair.

RMPL and Titan also offer many of the goal-directed tasking and monitoring capabilities of AI robotic execution languages, like ESL[17] and RAPS[18]. One key difference is that RMPL's constructs fully cover synchronous programming, hence moving towards a unification of a goal-directed AI executive with its underlying real-time language. In addition, Titan's deductive controller handles a rich set of system models, moving execution languages towards a unification with model-based autonomy.

8 Acknowledgments

This work was supported in part by the DARPA MOBIES program under contract F33615-00-C-1702, and by NASA's Cross Enterprise Technology Development program under contract NAG2-1466. The authors would like to thank Seung Chung, Paul Elliott, Lorraine Fesq, Vineet Gupta, Michael Pekala, Robert Ragno, Gregory Sullivan, John Van Eepoel, and David Watson.

References

1. Young, T., *et al.*: Report of the Mars Program Independent Assessment Team. NASA Tech. Rep. (March 2000)
2. Williams, B.C., Nayak, P.: A Model-Based Approach to Reactive Self-configuring Systems. In: Proceedings of AAAI-96, Vol. 2. AAAI Press, Menlo Park, CA, USA (1996) 971–978
3. Williams, B.C., Nayak, P.: A Reactive Planner for a Model-Based Executive. In: Proceedings of IJCAI-97, Vol. 2. Morgan Kaufmann, San Francisco, CA, USA (1997) 1178–1185
4. Ingham, M., Ragno, R., Williams, B.C.: A Reactive Model-Based Programming Language for Robotic Space Explorers. In: Proceedings of ISAIRAS-01. Montreal, Canada (2001)
5. Chung, S., Van Eepoel, J., Williams, B.C.: Improving Model-Based Mode Estimation Through Offline Compilation. In: Proceedings of ISAIRAS-01. Montreal, Canada (2001)
6. Kim, P., Williams, B.C., Abramson, M.: Executing Reactive Model-Based Programs Through Graph-Based Temporal Planning. In: Proceedings of IJCAI-01, Vol. 1. Morgan Kaufmann, San Francisco, CA, USA (2001) 487–493
7. Bernard, D., *et al.*: Design of the Remote Agent Experiment for Spacecraft Autonomy. In: Proceedings of the IEEE Aerospace Conference. Snowmass at Aspen, CO, USA (1999)
8. Ingham, M., *et al.*: Autonomous Sequencing and Model-Based Fault Protection for Space Interferometry. In: Proceedings of ISAIRAS-01. Montreal, Canada (2001)
9. Goodrich, C., Kurien, J., Continuous Measurements and Quantitative Constraints - Challenge Problems for Discrete Modeling Techniques. In: Proceedings of ISAIRAS-01. Montreal, Canada (2001)
10. Berry, G., Gonthier, G.: The Synchronous Programming Language ESTEREL: Design, Semantics, Implementation. Science of Computer Programming, Vol. 19, No.2. (1992) 87–152
11. Harel, D.: Statecharts: A Visual Formulation for Complex Systems. Science of Computer Programming, Vol. 8, No. 3. (1987) 231–274
12. Gupta, V., Jagadeesan, R., Saraswat, V.: Models of Concurrent Constraint Programming. In: Proceedings of the International Conference on Concurrency Theory (CONCUR 1996), Lecture Notes in Computer Science, Vol. 1119. Springer-Verlag, Berlin Heidelberg (1996) 66–83
13. Davis, R.: Diagnostic Reasoning Based on Structure and Behavior. Artificial Intelligence, Vol. 24. (1984) 347–410
14. Williams, B.C., Chung, S., Gupta, V.: Mode Estimation of Model-Based Programs: Monitoring Systems with Complex Behavior. In: Proceedings of IJCAI-01, Vol. 1. Morgan Kaufmann, San Francisco, CA, USA (2001) 579–590
15. Williams, B.C., Ragno, R.J.: Conflict-Directed A* and its Role in Model-Based Embedded Systems. To appear: Journal of Discrete Applied Mathematics.
16. Di Pierro, A., Wiklicky, H.: An Operational Semantics for Probabilistic Concurrent Constraint Programming. In: Proceedings of the International Conference on Computer Languages (ICCL 98). IEEE Computer Society Digital Library (1998) 174–183
17. Gat, E.: ESL: A Language for Supporting Robust Plan Execution in Embedded Autonomous Agents. In: Plan Execution: Problems and Issues. Papers from the 1996 Fall Symposium. AAAI Press, Menlo Park, CA, USA (1996) 59–64
18. Firby, R.J.: The RAP Language Manual. In: Animate Agent Project Working Note AAP-6, University of Chicago. Chicago, IL, USA (March 1995)