Model-Based Programming of Intelligent Embedded Systems and Robotic Space Explorers

BRIAN C. WILLIAMS, MEMBER, IEEE, MICHEL D. INGHAM, SEUNG H. CHUNG, AND PAUL H. ELLIOTT

Invited Paper

Programming complex embedded systems involves reasoning through intricate system interactions along lengthy paths between sensors, actuators, and control processors. This is a challenging, time-consuming, and error-prone process requiring significant interaction between engineers and software programmers. Furthermore, the resulting code generally lacks modularity and robustness in the presence of failure. Model-based programming addresses these limitations, allowing engineers to program reactive systems by specifying high-level control strategies and by assembling commonsense models of the system hardware and software. In executing a control strategy, model-based executives reason about the models "on the fly," to track system state, diagnose faults, and perform reconfigurations. This paper develops the Reactive Model-Based Programming Language (RMPL) and its executive, called Titan. RMPL provides the features of synchronous, reactive languages, with the added ability of reading and writing to state variables that are hidden within the physical plant being controlled. Titan executes an RMPL program using extensive component-based declarative models of the plant to track states, analyze anomalous situations, and generate novel control sequences. Within its reactive control loop, Titan employs propositional inference to deduce the system's current and desired states, and it employs model-based reactive planning to move the plant from the current to the desired state.

Keywords*—*Constraint programming, model-based autonomy, model-based execution, model-based programming, model-based reasoning, robotic execution, synchronous programming.

I. Introduction

Embedded systems, from automobiles to office-building control systems, are achieving unprecedented levels of robustness by dramatically increasing their use of computation. 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 may need to 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 development of these systems by creating embedded programming languages that are able to reason about and control underlying hardware from engineering models. We call this approach *model-based programming*.

A. Robustness in Deep Space

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 teams and by using many operations personnel to handle unforeseen circumstances as they arise. 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 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.

For example, consider the leading hypothesis for the cause of the Mars Polar Lander failure. 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,

Manuscript received December 20, 2001; revised August 31, 2002. This work was supported in part by the Defense Advanced Research Projects Agency Model-Based Integrated Embedded Software program under Contract F33615-00-C-1702 and in part by NASA's Cross Enterprise Technology Development prgram under Contract NAG2–1466

The authors are with the Space Systems and Artificial Intelligence Laboratories, Massachusetts Institute of Technology, Cambridge, MA 02139 USA (e-mail: williams@mit.edu; ingham@mit.edu; chung@mit.edu; pelliott@mit.edu).

Digital Object Identifier 10.1109/JPROC.2002.805828

which were designed to turn off the engine when triggered. 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 [2].

Suppose for a moment that the lander had been piloted by a human. The pilot would have received one altimeter reading of 40 m elevation, and a split second later a reading of touchdown from the leg sensors. It seems unlikely that the pilot would immediately respond by shutting down the engine, given how these observations defy our common sense about physics. Rather, the pilot would consider that a sensor failure was most likely, gather additional information to determine the correct altitude, and choose a conservative approach in the presence of uncertainty.

Of course, such embedded systems cannot be piloted by humans; they must be preprogrammed. However, the space of potential failures and their interactions with the embedded software is far too large for programmers to successfully enumerate, and correctly encode, within a traditional programming language.

Our objective is to support future programmers with embedded languages that avoid commonsense mistakes, by automatically reasoning from hardware models. Our solution to this challenge has two parts. First, we are creating increasingly intelligent embedded systems that automatically diagnose and plan courses of action at reactive time scales, based on models of themselves and their environment [3]–[7]. This paradigm, called *model-based autonomy*, has been demonstrated in space on the NASA Deep Space One (DS-1) probe [8], and on several subsequent space systems [9], [10]. 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 execution 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*.

B. Model-Based Programming

Engineers like to reason about embedded systems in terms of state evolutions. However, embedded programming languages, such as Esterel [11] and Statecharts [12], interact with a physical plant by reading sensors and setting control variables [Fig. 1 (left)]. It is then 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 large number of possible scenarios makes this an error-prone process.

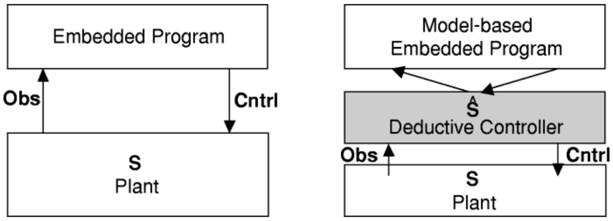


Fig. 1 Model of interaction with the physical plant for traditional embedded languages (left) and model-based programming (right).

A model-based programming language is similar to reactive embedded languages like Esterel, with the key difference that it interacts directly with the plant state [Fig. 1, right]. This is accomplished by allowing the programmer to *read or write "hidden" state variables in the plant*, that is, 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 control variables. This mapping is performed automatically by employing a deductive controller that reasons from a commonsense plant model.

A model-based program is composed of two components. The first is a *control program,* which uses standard programming constructs 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 component is a *plant model*, which captures the physical plant's nominal behavior and common failure modes. This model unifies constraints, concurrency, and Markov processes.

C. Model-Based Execution

A model-based program is executed by automatically generating a control sequence that moves the physical plant to the states specified by the program (see Fig. 2). We call these specified states *configuration goals*. Program execution is achieved using a *model-based executive*, such as Titan, which repeatedly generates the next configuration goal, and a sequence of control actions that achieve this goal, based on knowledge of the current plant state and plant model. The model-based executive continually estimates the most likely state of the plant from sensor information and the plant model. This information allows the executive to confirm the successful execution of commands and the achievement of configuration goals, and to diagnose failures.

A model-based executive continually tries to transition the plant toward a state that satisfies the configuration goals, while maximizing some reward metric. When the plant strays from the specified goals due to failures, the executive 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.

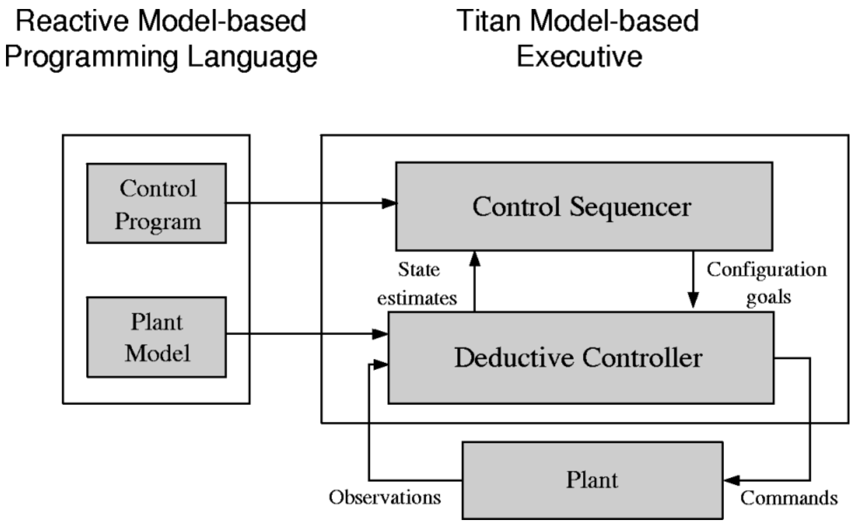


Fig. 2 Architecture for a model-based executive.

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 achieve. 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*).

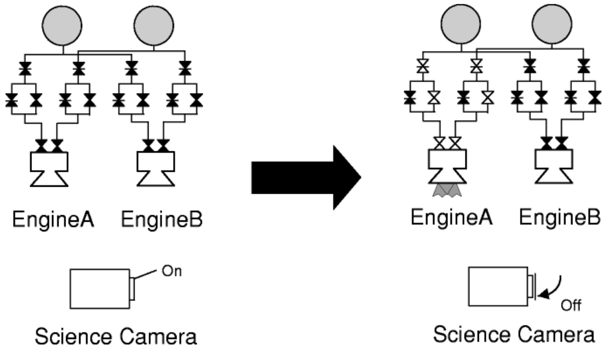


Fig. 3 Simple spacecraft for the orbital insertion scenario. Initial state (left) and goal state (right) are depicted.

D. Outline

In this paper we develop RMPL and its corresponding executive, *Titan*. Section II illustrates model-based programming in detail on a simple example. Section III specifies the formal semantics for a model-based program. Section IV introduces the constructs of RMPL required for specifying control behavior. The remaining sections develop Titan. Section V describes the control sequencer, which translates the RMPL control program into a sequence of state configuration goals, based on the plant's estimated state trajectory. Section VI describes the deductive controller, which uses a commonsense plant model to estimate the state of the system and generate control actions that achieve the state configuration goals provided by the sequencer. Section VII concludes the paper with a discussion of related work.

II. 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 Fig. 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 engine 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 engines shown in the figure. 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 novel failures as they arise. This is essential for achieving high levels of robustness.

As an example, consider the model-based program corresponding to the previously described specification for spacecraft orbital insertion. The dual main engine system (see Fig. 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.

We start by specifying the two components of a model-based program for orbital insertion: the control program and plant model. We then describe the execution of the program under nominal and failure situations.

A. Control Program

The RMPL control program, shown in Fig. 4, encodes the informal specification we gave previously as a set of state trajectories. The specific RMPL constructs used in the program are introduced in Section IV. Recall that to perform orbital insertion, one of the two engines must be fired. We start by concurrently placing the two engines in the standby state and by shutting off the camera. This is performed by lines 3–5, where commas at the end of each line denote 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).

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.

B. Plant Model

The plant model is used by a model-based executive to map queried and asserted states in the control program to sensed variables and control sequences, respectively, in the physical plant. The 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 probabilistic transitions between modes. The component automata operate concurrently and synchronously. In Section III, we describe the semantics of plant models in terms of partially observable Markov decision processes.

For the orbital insertion example, we can model the spacecraft abstractly as a three-component system (two engines and a camera) by supplying the models depicted graphically in Fig. 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 Fig. 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 that the engine may fail in a way never seen before, by specifying no constraints for the engine's behavior in the $failed$ mode. This approach, called *constraint suspension* [13], is common to most approaches to model-based diagnosis.

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

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

Models include commanded and uncommanded transitions, both of which are probabilistic. For example, the engine has uncommanded transitions from off, $standby$, and $firing$ to $failed$. The transitions have a 1% probability, and are shown in the figure as arcs labeled 0.01. Transitions between nominal modes are triggered on commands, and occur with probability 99%.

C. 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 variable assignments: "EngineA = Standby," "EngineB = Standby," and "Camera = Off" (lines 3–5, Fig. 4). 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, Fig. 4). 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

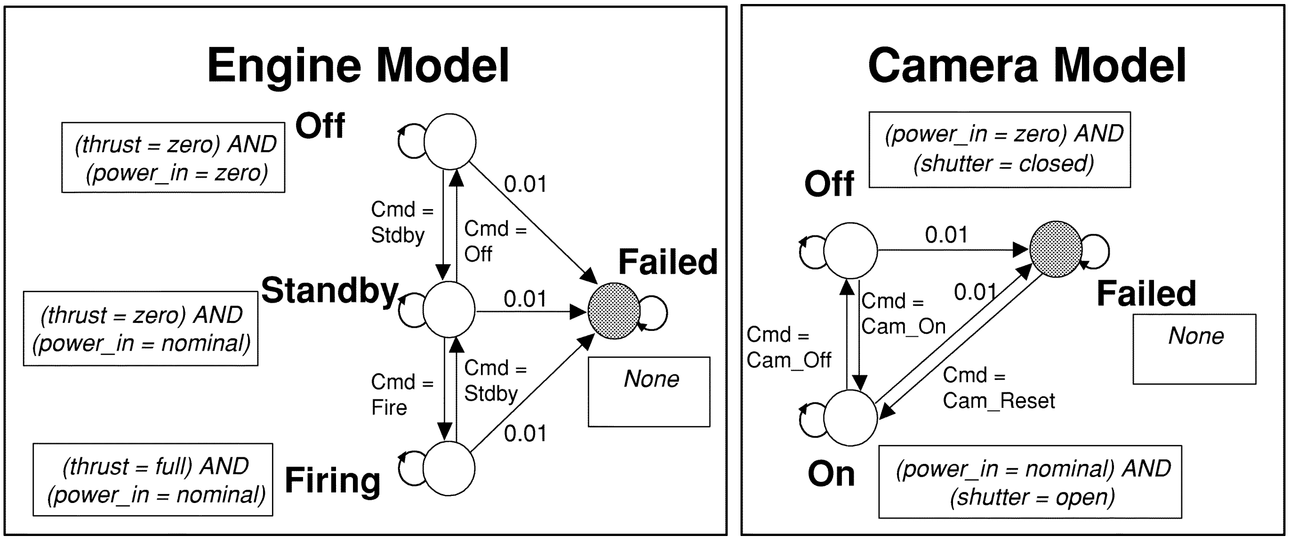

Engine Model
Off
(thrust = zero) AND (power_in = zero)
Cmd = Stdby
Cmd = Off
0.01
Failed
Standby
(thrust = zero) AND (power_in = nominal)
0.01
None
Cmd = Fire
Cmd = Stdby
0.01
Firing
(thrust = full) AND (power_in = nominal)
Camera Model
(power_in = zero) AND (shutter = closed)
Off
0.01
Failed
Cmd = Cam_On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

Fig. 5 State transition models for a simplified spacecraft. The probabilities on nominal transitions are omitted for clarity.

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, Fig. 4), which is issued by the control sequencer to the deductive controller.

III. Model-Based Program Execution Semantics

Next, we define the execution of a model-based program in terms of legal state evolutions of a physical plant, and define the functions of the control sequencer and deductive controller (see Fig. 2).

A. Plant Model

A plant is modeled as a *partially observable Markov decision process* (POMDP) $\mathcal{P} = \langle \Pi, \Sigma, \mathbb{T}, \mathbf{P}_\Theta, \mathbf{P}_\mathbb{T}, \mathbf{P}_\mathbb{O}, \mathbb{R} \rangle$. Π is a set of *variables*, each ranging over a finite domain. Π is partitioned into *state variables* Π^s, *control variables* Π^c, and *observable variables* Π^o. 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. An *observation* of the plant, o, is defined as an assignment to each variable in Π^o. A *control action*, μ, is defined as an assignment to each variable in Π^c.

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

A *plant trajectory* is a (finite or infinite) sequence of feasible states $[s^{(0)}, s^{(1)}, \ldots]$, such that for each $s^{(t)}$ there is a feasible assignment $\sigma^{(t)} \in \Sigma$ that agrees with $s^{(t)}$ on assignments to variables in Π^s, and $s^{(t+1)} = \tau(\sigma^{(t)})$, 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.

The space of possible state trajectories for a plant can be visualized using a *Trellis diagram*, which enumerates all possible states at each time step and all transitions between states at adjacent times [see Fig. 6(a)].

B. Model-Based Program Execution

A model-based program consists of a plant model $\mathcal{P}$, described previously, and a control program $\mathcal{CP}$, described as a deterministic automaton $\mathcal{CP} = \langle \Sigma_{\rm cp}, \theta_{\rm cp}, \tau_{\rm cp}, g_{\rm cp}, \Sigma_s \rangle$. $\Sigma_{\rm cp}$ is the set of *program locations*, where $\theta_{\rm cp} \in \Sigma_{\rm cp}$ is the program's initial location. A program location represents the "state" of the program's execution at any given time. Transitions $\tau_{\rm cp}$ between locations are conditioned on plant states Σ_s of $\mathcal{P}$; that is, $\tau_{\rm cp}$ is a function $\tau_{\rm cp} : \Sigma_{\rm cp} \times \Sigma_s \rightarrow \Sigma_{\rm cp}$. Each location $l \in \Sigma_{\rm cp}$ has a corresponding *configuration goal* $g_{\rm cp}(l) \subset \Sigma_s$, which is the set of legal plant goal states associated with location l.

A *legal execution* of a model-based program is a trajectory of feasible plant state estimates, $[\hat{s}^{(0)}, \hat{s}^{(1)}, \ldots]$ of $\mathcal{P}$, and locations $[l^{(0)}, l^{(1)}, \ldots]$ of $\mathcal{CP}$ such that: 1) $\hat{s}^{(0)}$ is a valid initial plant state, that is, $\mathbf{P}_\Theta(\hat{s}^{(0)}) > 0$; 2) $\hat{s}^{(t)}$ is consistent with the observations and the plant model, that is, for each $\hat{s}^{(t)}$, there is a $\sigma^{(t)} \in \Sigma$ of $\mathcal{P}$ that agrees with $\hat{s}^{(t)}$ and $o^{(t)}$, on the corresponding subsets of variables; 3) $l^{(0)}$ is the initial program location $\theta_{\rm cp}$; 4) $\langle l^{(t)}, l^{(t+1)} \rangle$ represents a legal control program transition, that is, $l^{(t+1)} = \tau_{\rm cp}(l^{(t)}, \hat{s}^{(t)})$; and 5) if plant state $\hat{s}^{(t+1)}$ is the result of a nominal plant transition from $\sigma^{(t)}$, that is, $\hat{s}^{(t+1)} = \tau^n(\sigma^{(t)})$, then either $\hat{s}^{(t+1)}$ is the maximum-reward state in $g_{\rm cp}(l^{(t)})$, or $\langle \hat{s}^{(t)}, \hat{s}^{(t+1)} \rangle$ is the prefix of a simple nominal plant trajectory that ends in the maximum-reward state in $g_{\rm cp}(l^{(t)})$.

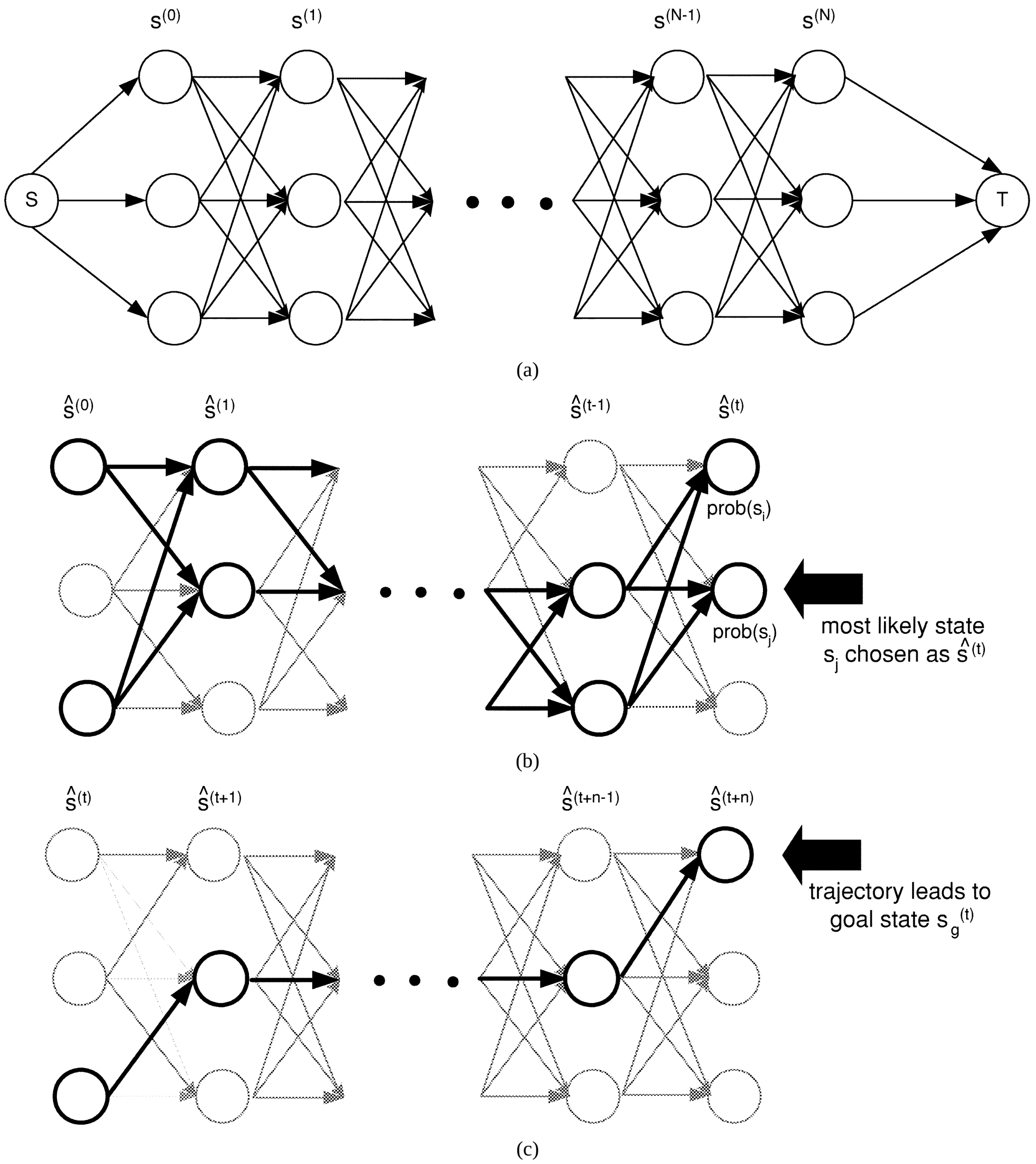


Fig. 6 (a) A Trellis diagram depicting the system's possible state trajectories. (b) Mode estimation tracks a set of trajectories through the diagram, and selects the most likely state as its estimate for $\hat{s}^{(t)}$. (c) Mode reconfiguration chooses a path through the diagram along nominal transitions, terminating at the goal state $s_g^{(t)}$.

C. 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 input, a control program $\mathcal{CP}$, and a sequence $[\hat{s}^{(0)}, \hat{s}^{(1)}, \ldots]$ of plant state estimates. It generates a sequence $[g^{(0)}, g^{(1)}, \ldots]$ of configuration goals.

A deductive controller takes, as input, the plant model $\mathcal{P}$, a sequence of configuration goals $[g^{(0)}, g^{(1)}, \ldots]$, and a sequence of observations $[o^{(0)}, o^{(1)}, \ldots]$. It generates a sequence of most likely plant state estimates $[\hat{s}^{(0)}, \hat{s}^{(1)}, \ldots]$ and a sequence of control actions $[\mu^{(0)}, \mu^{(1)}, \ldots]$.

The sequence of state estimates is generated by a process called *mode estimation* (ME). ME is an online algorithm for tracking the most likely states that are consistent with the plant model, the sequence of observations and the control actions. ME is framed as an instance of Hidden Markov Model belief state update, which computes the probability associated with being in each state s_j at time $t+1$, according to the following equations:

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

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

where $\mathbf{P}_{\mathbb{T}}(\sigma_i, s_j)$ is defined as the probability that $\mathcal{P}$ transitions from σ_i to state s_j, computed as the sum of $\mathbf{P}_{\mathcal{T}}$ over all transition functions τ that map σ_i to state s_j. The prior probability $p^{(\bullet t+1)}[s_j]$ is conditioned on all observations up to $o^{(t)}$, while the posterior probability $p^{(t+1\bullet)}[s_j]$ is also conditioned on the latest observation $o^{(t+1)} = o_k$.

Belief state update associates a probability to each state in the Trellis diagram. For mode estimation, the tracked state with the highest belief state probability is selected as the most likely state $\hat{s}^{(t)}$, as shown in Fig. 6(b).

The sequence of control actions is generated by a process called *mode reconfiguration* (MR). MR takes as input a configuration goal $g^{(t)}$ and the most likely current state $\hat{s}^{(t)}$ from ME, and it returns a series of commands that progress the plant toward a maximum-reward goal state that achieves $g^{(t)}$. MR can be thought of as picking a simple path through the Trellis diagram along nominal transitions that leads to the goal state, as shown in Fig. 6(c).

MR computes a goal state $s_g^{(t)}$ associated with configuration goal $g^{(t)}$, and a control action $\mu^{(t)}$, such that: 1) $s_g^{(t)}$ is the state that entails the configuration goal $g^{(t)}$, while maximizing reward $\mathbb{R}$; 2) $s_g^{(t)}$ is reachable from $\hat{s}^{(t)}$ through a sequence of nominal transitions of τ^n; and 3) $\mu^{(t)}$ is a control action that transitions the plant state from $\hat{s}^{(t)}$ to $\hat{s}^{(t+1)}$, where either $\hat{s}^{(t+1)} = s_g^{(t)}$ or $\langle \hat{s}^{(t)}, \hat{s}^{(t+1)} \rangle$ is the prefix of a simple nominal state trajectory that leads to state $s_g^{(t)}$.

Just as concurrent constraint programming offers a family of languages, each characterized by a choice of constraint system, model-based programming defines a family of languages, each characterized by the choice of the underlying plant modeling formalism. In the following three sections of the paper, we define one particularly powerful instance of a model-based programming language and its corresponding executive. The practical importance of this instance has been demonstrated through deployments on a wide range of applications from the automotive and aerospace domains [14]. In the next section, we define RMPL. In the subsequent two sections, we present in more detail the computational models and algorithms used by Titan's control sequencer and deductive controller implementations.

IV. The Reactive Model-Based Programming Language

In this section we introduce RMPL, by first introducing its underlying constraint system, and then developing constructs for writing control programs.

A. Constraint System: Propositional State Logic

A constraint system $(D, \models)$ 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 [15]. RMPL currently supports *propositional state logic* as its constraint system. In propositional state logic, a *proposition* is, in general, an assignment $(x = v)$, where variable x ranges over a finite domain $\mathbb{D}(x)$. A proposition can have a truth assignment of true or false. Propositions are composed into formula 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*. We denote entailment by simply stating the constraint. Nonentailment is denoted by using an overbar. Note that nonentailment of constraint c (denoted $\overline{c}$) 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 Esterel [11] and the Timed Concurrent Constraint (TCC) programming language [16], 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 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.[1]

B. RMPL Control Programs

To motivate RMPL's constructs we consider a more complex example, taken from the NASA DS-1 probe. DS-1 uses an ion propulsion engine, which thrusts almost continuously throughout the mission. Once a week the engine is turned off, in order for DS-1 to perform a course correction, called *optical navigation*. The RMPL control program for optical navigation (OpNav) is shown in Fig. 7. OpNav works by taking pictures of three asteroids and by using the difference between actual and projected locations to determine the course error. OpNav first shuts down the ion engine and prepares its camera concurrently. It then uses attitude control thrusters to turn toward each of three asteroids, uses the camera to take a picture of each, and stores each picture in memory. The three images are then read and processed, and a course correction is computed. One of the more subtle failures that OpNav may experience is an undetected fault in a navigation image. If the camera generates a faulty image, it gets stored in memory. After all three images are stored, the images are processed, and only then is the problem detected.

1) Desiderata: OpNav highlights five design features for RMPL. First, the program exploits full concurrency, by intermingling sequential and parallel threads of execution. For example, the camera is turned on and the engine is set to standby in parallel (lines 3–4), while pictures are taken serially (lines 7–9). Second, it involves conditional execution, such as switching to standby if the engine is firing (line 4).

[1] Note that, although RMPL provides the flexibility of asserting any form of constraint on plant variables, the current implementation of the Titan model-based executive handles only assertions of constraints that are conjunctions of state variable assignments.

```
1   OpNav() :: {
2     do {
3       Camera = On,
4       if Engine = Firing  thennext Engine = Standby,
5       do {
6         when (Engine = Off ∨ Engine = Standby) ∧ Camera = On  donext {
7           TakePicture(Asteroid1);
8           TakePicture(Asteroid2);
9           TakePicture(Asteroid3);
10          {
11            Camera = Off,
12            ComputeCorrection()
13          }
14        }
15      }  watching SnapStoreStatus = Failed,
16      when CorrectionStatus = Succeeded  donext OpNavStatus = Succeeded,
17      when SnapStoreStatus = Failed ∨ CorrectionStatus = Failed  donext
18          OpNavStatus = Failed
19    }  watching OpNavStatus = Succeeded ∨ OpNavStatus = Failed
20  }
21
22  TakePicture(target) :: {
23    do {
24      Attitude = target,
25      when Attitude = target  donext {
26        SnapStore();
27        SnapStore();
28        SnapStoreStatus = Failed
29      }
30    }  watching Picture = Stored
31  }
32
33  SnapStore() ::{
34    do {
35      do Picture = Taken  watching Camera = Failed,
36      do {
37        whenever Camera = Failed  donext {
38          Camera = On;
39          do Picture = Taken  watching Camera = Failed
40        }
41      }  watching Picture = Taken,
42      when Picture = Taken  donext  Picture = Stored
43    }  watching Picture = Corrupted
44  }
45
46  ComputeCorrection():: {
47    do {
48      Correction = Computed
49    }  watching Correction = Failed;
50    if Correction = Computed  thennext  CorrectionStatus = Succeeded
51       elsenext  CorrectionStatus = Failed
52  }
```

Fig. 7 RMPL control program for the DS-1 optical navigation procedure. In RMPL programs, comma delimits parallel processes and semicolon delimits sequential processes.

Third, it involves iteration; for example, "**when** (Engine = Off ∨ Engine = Standby) ∧ Camera = On **donext** … " (line 6) says to iteratively test until the engine is either off or in standby and the camera is on, and then to proceed. Fourth, the program involves preemption; for example, "**do** … **watching** SnapStoreStatus = Failed" (lines 5–15) says to perfom a task, but to interrupt it as soon as the watched condition (SnapStoreStatus = Failed) is entailed. Procedures used by OpNav, such as TakePicture, exploit similar features. These four features are common to most synchronous reactive programming languages. As highlighted in the preceding sections, the fifth and defining feature of RMPL is the ability to reference hidden states of the physical plant within assertions and guards, such as "**if** Engine = Firing **thennext** Engine = Standby" (line 4), where constraints on the hidden state of the engine are used in both a condition check and an assertion.

RMPL is an object-oriented, constraint-based language in a style similar to Java. We develop RMPL by first introducing a small set of primitives for constructing model-based control programs. These primitives are fully orthogonal, that is, they may be nested and combined arbitrarily. The RMPL primitives are closely related to the TCC programming language [16]. To make the language usable, we define on top of these primitives a variety of derived combinators, such as those used in the orbital insertion and optical navigation control programs (see Fig. 4 and 7).

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

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

where an assertion is an *achieve* constraint, made up of conjunctions of propositions. Note that we allow procedure calls specified as RMPL program invocations, in which program_name corresponds to another specified 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. For example, the OpNav control program calls the procedure TakePicture three times, each time with a different argument (Asteroid1, Asteroid2, and Asteroid3). With each invocation of TakePicture, the argument replaces the parameter "target," resulting in a specific Attitude variable assertion (Asteroid1, Asteroid2, and Asteroid3 are all members in the domain of the Attitude state variable).

2) 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 they act on the plant state in the next time instant.

g: This expression asserts that the plant should progress toward a state that entails constraint g. g is an *achieve* constraint on variables of the physical plant. This is the basic construct for affecting the plant's hidden state.

A **maintaining** *c:* This expression executes expression A, while ensuring that constraint c is maintained true throughout. c is an *ask* constraint on 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 nonentailment.

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 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 the variables of the physical plant. This is the basic construct for conditionally branching upon entailment of the plant's hidden state.

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 the variables of the physical plant. This is the basic construct for conditionally branching upon nonentailment of the plant's hidden state.

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**.

The previously described primitive combinators cover the five desired design features. We can use them to implement a rich set of derived combinators, some of which are used in the orbital insertion and optical navigation examples. For example, the derived construct **when** c **donext** A 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. The set of derived combinators is included as an Appendix to this paper.

The primitive and derived RMPL constructs are used to encode control programs. This subset is sufficient to implement most of the control constructs of the Esterel language [11]. A mapping between key Esterel constructs and analogous expressions in RMPL was presented in [5]. 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 [17]. The plant model is further defined in Section VI.

V. Control Sequencer

The RMPL control program is executed by the 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 as a specific instance of the deterministic control program automaton presented in Section III. In addition, we discuss the compilation from RMPL to HCA, and we present the execution algorithm used by Titan's control sequencer.

A. Hierarchical Constraint Automata

To efficiently execute RMPL programs, we translate each of the primitive combinators, introduced in the previous section, into an HCA. In the following we call the "states" of an HCA *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" HCA locations. 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 toward, 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 marked. 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 [11] and State Charts [12]. 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* $\mathcal{A}$ is a tuple $\langle \Sigma, \Theta, \Pi, \mathbb{G}, \mathbb{M}, \mathbb{T} \rangle$, where:

1) 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. We define the set of *subautomata* of $\mathcal{A}$ as the set of locations of $\mathcal{A}$, and locations that are descendants of the composite locations of $\mathcal{A}$, that is, the subautomata of $\mathcal{A}$ are given by the following recursive function:

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

2) $\Theta \subseteq \Sigma$ is the set of $\mathcal{A}$'s *start locations* (also called the *initial marking* of $\mathcal{A}$).

3) Π is the set of plant state variables, with each $x \in \Pi$ ranging over a finite domain $\mathbb{D}[x]$. $\mathbb{C}[\Pi]$ denotes the set of all finite-domain constraints over variables in Π, and $\mathbb{C}_{ach}[\Pi]$ denotes the set of finite-domain *achieve* constraints over Π, where an *achieve* constraint is a conjunction of state variable assignments.

4) $\mathbb{G} : \Sigma_p \to \mathbb{C}_{ach}[\Pi]$ associates with each primitive location $\sigma^p \in \Sigma_p$ a finite-domain constraint $\mathbb{G}(\sigma^p)$ that the plant progresses toward whenever σ^p is marked. $\mathbb{G}(\sigma^p)$ is called the *goal constraint* of σ^p. Goal constraints $\mathbb{G}(\sigma^p)$ may be thought of as abstract set points, representing a set of states that the plant must evolve toward when σ^p is marked.

5) $\mathbb{M} : \Sigma \to \mathbb{C}[\Pi]$, associates with each location $\sigma \in \Sigma$ a finite-domain constraint $\mathbb{M}(\sigma)$ that must hold at the current instant for σ to be marked. $\mathbb{M}(\sigma)$ is called the *maintenance constraint* of σ. Maintenance constraints $\mathbb{M}(\sigma)$ may be viewed as representing monitored constraints that must be maintained in order for execution to progress toward achieving any goal constraints specified within σ.

6) $\mathbb{T} : \Sigma \times \mathbb{C}[\Pi] \to 2^{\Sigma}$ associates with each location $\sigma \in \Sigma$ a transition function $\mathbb{T}(\sigma)$. Each $\mathbb{T}(\sigma) : \mathbb{C}[\Pi] \to 2^{\Sigma}$

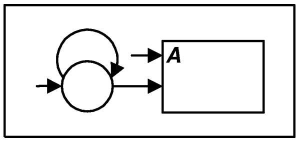


Fig. 8 HCA representation of the RMPL always construct.

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*. $\mathcal{M}$ denotes the set of possible markings, where $\mathcal{M} \subseteq 2^{\Sigma}$.

As an example of an HCA, consider the RMPL combinator **always** A, which maps to the graphical representation in Fig. 8. This automaton results in the start locations of A being marked at each time instant. The locations, Σ, of the automaton consist of primitive location σ_{new}, drawn to the left as a circle, and composite location A, drawn to the right as a rectangle. The start locations Θ are σ_{new} and A, and are indicated by two short arrows.

In the graphical representation of HCA, primitive locations are represented as circles, while composite locations are represented as rectangles. Constraints are indicated by lowercase letters, such as c, g, or m. 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**. In the previous example, σ_{new} implicitly has goal and maintenance constraints **True**; other constraints may be specified within A.

Transitions are conditioned on constraints that must be satisfied 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, 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 $\overline{c}$), 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' are the source and target of an arc with label l. Again, if no label is indicated, it is implicitly **True**. The HCA representation of **True** always A in Fig. 8 has two transitions, one from σ_{new} to itself and one from σ_{new} to A. Both these transitions' labels are implicitly **True**.

Our HCA encoding has four properties that distinguish it from the hierarchical automata employed by other reactive embedded languages [11], [12], [18]. 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 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 can be enabled based on lack of information, that is, nonentailment 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.

It should be noted that two extensions to HCA have been presented elsewhere. In [17], we consider the problem of the state estimation of probabilistic variants of HCA. In [7], we consider the execution of model-based programs encoded as timed variants of HCA that allow for representation of nondeterministic choice.

B. Executing HCA

Informally, execution of an HCA proceeds as follows. The control sequencer begins with a marked subset of the HCA's locations and an estimate of the current plant state from mode estimation. It then creates a set consisting of each marked location whose maintenance constraint is satisfied by the current estimated state. Next, it conjoins all goal constraints of this set to produce a *configuration goal*. A configuration goal represents a set of states, such that the plant must progress toward one of these states, called the *goal state*, which has the greatest reward.[2] This configuration goal is then passed to mode reconfiguration, which executes a single command that makes progress toward achieving the goal. Next, the sequencer receives an update of the plant state from mode estimation. Based on this new state information, the HCA advances to a new marking, by taking all enabled transitions from marked primitive locations whose goal constraints are achieved or whose maintenance constraints have been violated, and from marked composite locations that no longer contain any marked subautomata. Finally, the cycle repeats.

More precisely, to execute an HCA $\mathcal{A}$, the control sequencer starts with an estimate of the current state of the plant, $\hat{s}^{(0)}$. It initializes $\mathcal{A}$ using $m_F(\mathcal{A})$, a function that marks the start locations of $\mathcal{A}$ and all their starting subautomata. It then repeatedly steps automaton $\mathcal{A}$ using the function Step_{HCA}, which maps the current state estimate and marking to a next marking and configuration goal. The functions m_F and Step_{HCA} are defined later. Execution "completes" when no marks remain, since the empty marking is a fixed point.

Given an HCA $\mathcal{A}$ 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 the following start locations:

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

For example, applying m_F to automaton **always** A returns the set consisting of **always** A, σ_{new}, A and any start locations contained within A.

Step_{HCA} transitions an automaton $\mathcal{A}$ from the current full marking to the next full marking, based on the current

[2] We define "progress" as taking an action that is part of a sequence of actions that leads to the goal state.

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

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 $\hat{s}^{(t)}$.
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 $\hat{s}^{(t)}$.
3) **Take action.** Given goal $g^{(t)}$ and state estimate $\hat{s}^{(t)}$, request that mode reconfiguration issue a command that progresses the plant towards a state 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 $\hat{s}^{(t+1)}$.
5) **Await incomplete goals.** If the goal constraint of a primitive location marked in $m^{(t)}$ is not entailed by $\hat{s}^{(t+1)}$, and its maintenance constraint was not violated by $\hat{s}^{(t)}$, then include that location as marked in $m^{(t+1)}$.
6) **Identify enabled transitions.** A transition from a marked primitive location σ^p in $m^{(t)}$ is enabled if both of the following conditions hold true:
    a) σ^p's goal constraint is satisfied by $\hat{s}^{(t+1)}$, or its maintenance constraint was violated by $\hat{s}^{(t)}$;
    b) the transition's guard condition is satisfied by $\hat{s}^{(t+1)}$.

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

Fig. 9 Step$_{\mathrm{HCA}}$ algorithm.

state estimate, and generates a new configuration goal. The Step$_{\mathrm{HCA}}$ algorithm is given in Fig. 9.

A *trajectory* of a hierarchical constraint automaton $\mathcal{A}$, given estimated plant state sequence $[\hat{s}^{(0)}, \hat{s}^{(1)}, \ldots]$, is a sequence of markings $[m^{(0)}, m^{(1)}, \ldots]$ and configuration goals $[g^{(0)}, g^{(1)}, \ldots]$ such that: 1) $m^{(0)}$ is the initial marking $m_F(\mathcal{A})$; and 2) for each $t \geq 0$, $\langle g^{(t)}, m^{(t+1)}, \hat{s}^{(t+1)} \rangle = \mathrm{Step}_{\mathrm{HCA}}(\mathcal{A}, m^{(t)}, \hat{s}^{(t)})$.

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

As an example, applying Step$_{\mathrm{HCA}}$ to the initial marking of **always** A causes σ_{new} to transition to A and back to σ_{new}, and for A to transition internally. σ_{new}'s transition back to itself ensures that it always remains marked. Its transition to A puts a new mark on A, initializing the starting subautomata of A at each time step. The ability of an HCA to have multiple locations marked simultaneously is key to the compactness of this novel encoding, by avoiding the need for explicit copies of A.

This example does not demonstrate the interaction with the physical plant through goal and maintenance constraints. We demonstrate this by revisiting the orbital insertion example in Section V-D.

C. Compiling RMPL to HCA

Each RMPL construct maps to an HCA as shown in Fig. 10. The derived combinators are definable in terms of the primitives, but for efficiency we map them directly to HCA 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 an HCA.

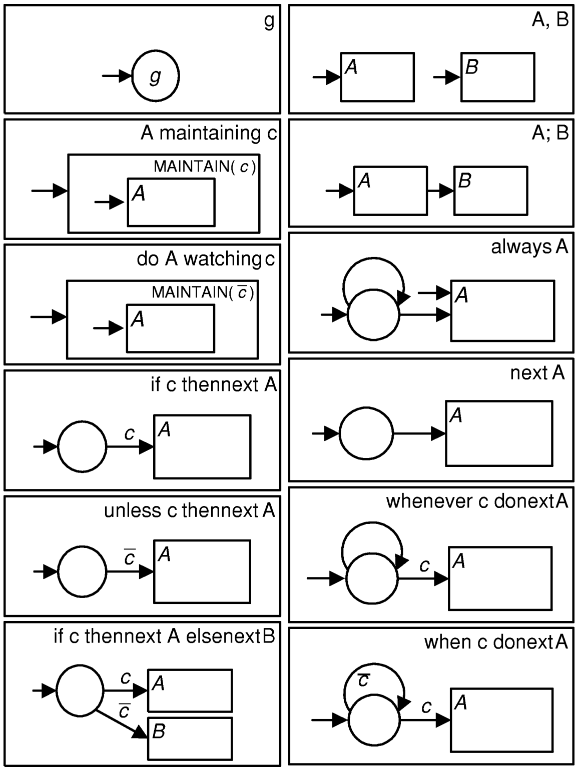


Fig. 10 Corresponding HCA for various RMPL constructs.

To illustrate compilation, consider the RMPL code for orbital insertion, shown earlier in Fig. 4. The RMPL compiler converts this to the corresponding HCA, shown in Fig. 11.[3]

D. Example: Executing the Orbital Insertion Control Program

The control sequencer interacts tightly with the mode estimation and mode reconfiguration capabilities of the deductive controller. As a practical example, we consider a nominal (i.e., failure-free) execution trace for the orbital insertion scenario. Markings are represented in the following figures by filling in the corresponding primitive locations. Any composite location with marked subautomata is considered marked. Locations are numbered 1–9 in the figures for reference.

Initial State: Initially, all start locations are marked (locations 1, 2, 3, 4, 5, 6, and 8 in Fig. 12). We assume mode estimation provides the following initial plant state estimate: {EngineA = Off, EngineB = Off, Camera = On}.

Execution will continue as long as the maintenance constraint on automaton 1, $\not\models$ (EngineA = Firing $\vee$ EngineB = Firing), remains true. In other words, execution of automaton 1 will terminate as soon as (EngineA = Firing $\vee$ EngineB = Firing) is entailed. Similarly, execution of automaton 5 will terminate if ever (EngineA = Failed) is entailed.

First Step: Since none of the maintenance constraints are violated for the initial state estimate, all start loca-

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

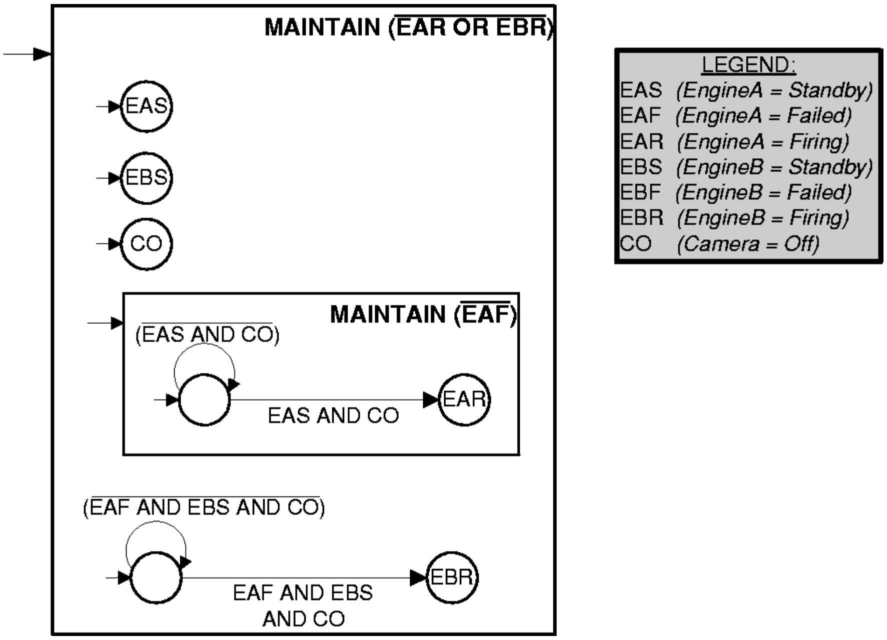


Fig. 11 HCA model for the orbital insertion scenario.

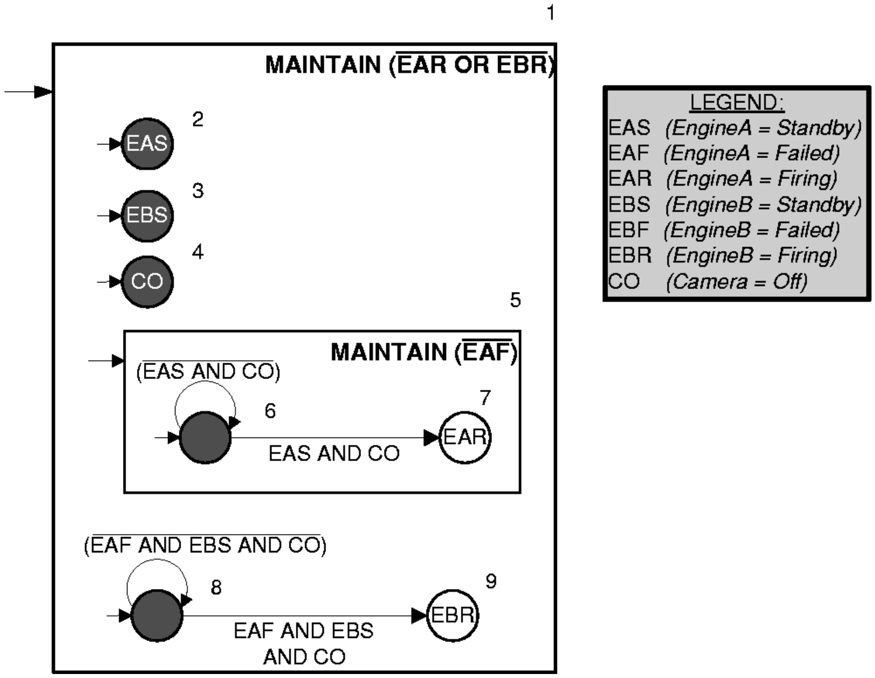


Fig. 12 Initial marking of the HCA for orbital insertion.

tions remain marked. The goal constraints asserted by the start locations consist of the following state assignments: (EngineA = Standby), (EngineB = Standby), and (Camera = Off) (from states 2, 3, and 4, respectively). These state assignments are conjoined into a configuration goal, and passed to mode reconfiguration. Mode reconfiguration then issues the first command in a command sequence that achieves the configuration goal [e.g., (Cmd = Cam_Off)].

In this example, mode estimation confirms that (Camera = Off) is achieved with a one-step operation, by observing that the shutter position sensor reads closed.

Locations 2 and 3, which assert (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 location 4, this thread of execution terminates. Marked locations 6 and 8, which correspond to "**when … donext …**" expressions in the RMPL control program, 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 is shown in Fig. 13.

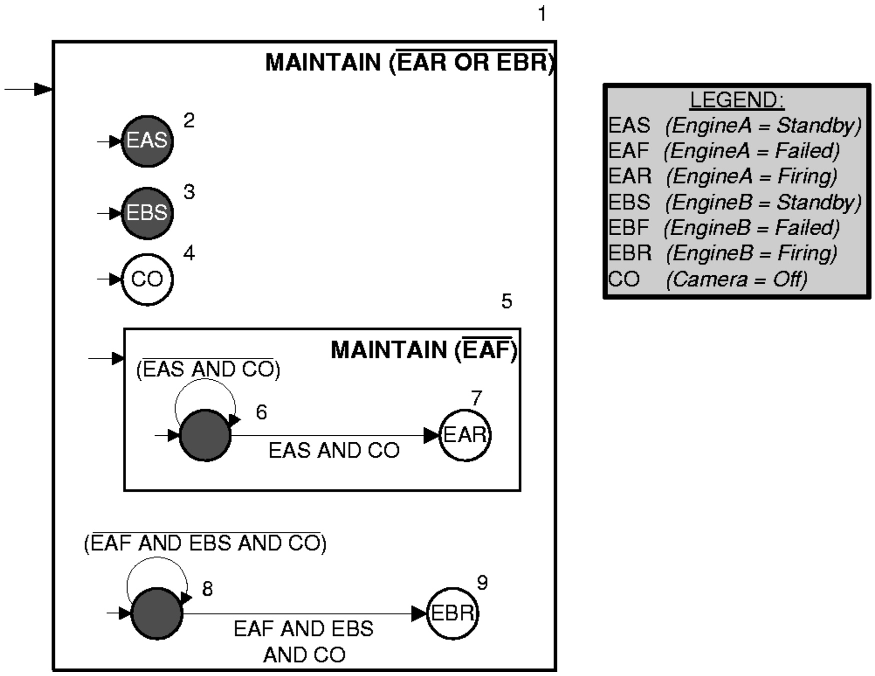


Fig. 13 Marking of the orbital insertion HCA after the first transition.

Second Step: The maintenance constraints corresponding to nonentailment of (EngineA = Failed) on automaton 5, and nonentailment of (EngineA = Firing $\vee$ EngineB = Firing) on automaton 1, still hold for the current state estimate, so all marked locations remain marked. Next, the goal constraints of the marked locations are collected, (EngineA = Standby $\wedge$ EngineB = Standby), and passed as a configuration goal to mode reconfiguration. Mode reconfiguration issues the first command in the sequence for achieving (EngineA = Standby) and (EngineB = Standby).

For the purposes of this example, we 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 constraints, at locations 2 and 3. In addition, for marked location 6, the transition labeled with condition (EngineA = Standby $\wedge$ Camera = Off) is enabled, and hence traversed to location 7. Finally, location 8 remains marked.

Thus, after taking the enabled transitions, only two primitive locations are marked, 7 and 8. This new marking is shown in Fig. 14.

Third Step: Since none of the maintenance constraints are violated for the current state estimate, all marked locations remain marked. Marked location 7 asserts the goal constraint (EngineA = Firing), which is passed to mode reconfiguration as the configuration goal. Mode reconfiguration determines that the engine fires by opening the valves that supply fuel and oxidizer, and issues the first command in a sequence that achieves this configuration.

We assume that this first command for achieving (EngineA = Firing) is executed correctly. Since the single goal constraint is not yet satisfied, and the only enabled transition is the self-transition at location 8, the markings and configuration goals remain unchanged (see Fig. 14).

Remaining Steps: Given the same configuration goal as the last step, mode reconfiguration issues the next command required to achieve (EngineA = Firing). It repeats this process until a flow of fuel and oxidizer are established, and mode estimation confirms that the engine is indeed firing. Since this violates the maintenance condition on automaton 1, the entire block of Fig. 11 is exited. Note that since no engine failure occurred during this process, the thread of execution waiting for (EngineA = Failed) did not advance from its start location (location 8).

VI. Deductive Controller

In this section, we present Titan's model-based deductive controller (see Fig. 15), which uses a plant model to estimate and control the state of the plant. A physical plant is composed of discrete and analog hardware and software. We model the behavior of the plant as a POMDP, which is encoded compactly using probabilistic *concurrent constraint automata (CCA).* Concurrency is used to model the behavior of a set of components that operate synchronously. Constraints are used to represent cotemporal interactions between state variables and intercommunication 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.

This section begins by defining CCA as a composition of constraint automata for individual components of a plant. The CCA encoding of the plant model is essential to the efficient operation of the deductive controller (see Fig. 15). In Section VI-E, we develop mode estimation, which estimates the most likely state of a CCA, and in Section VI-F,

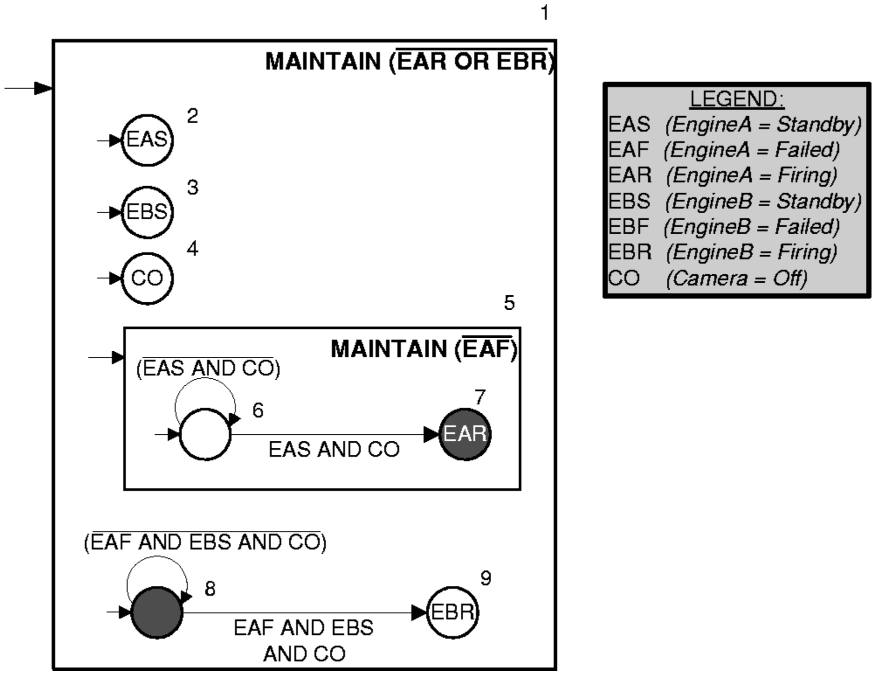


Fig. 14 Marking of the orbital insertion HCA after the second transition.

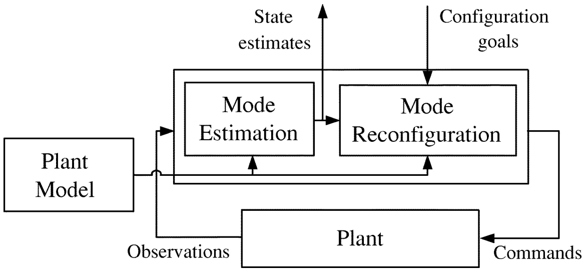


Fig. 15 Architecture for the deductive controller.

we develop mode reconfiguration, which generates a control sequence that transitions the CCA to a state that achieves a configuration goal.

A. Constraint Automata

The semantics of the plant model was introduced in Section III-A. Our model of a physical plant is encoded as a set of concurrently operating *constraint automata*, one constraint automaton for each component in the model. Constraint automata are component transition systems that communicate through shared variables. The transitions of each constraint automaton represent a rich set of component behaviors, including normal operation, failure, repair actions, and intermittency. The constraint automata operate synchronously; that is, at each time step every component performs a single state transition. Constraint automata for the simplified spacecraft components were provided pictorially in Fig. 5.

Each constraint automaton has an associated *mode variable* x_i with domain $\mathbb{D}(x_i)$. Given a current mode assignment, $x_i = v$, the component changes its mode by selecting a transition function $\tau_i(x_i = v)$ among a set of possible transition functions $\mathbb{T}_i(x_i = v)$, according to probability distribution $\mathbf{P}_{\mathbb{T}_i}(x_i = v)$. A component's behavior within each mode is modeled by an associated constraint $\mathbb{M}_i(x_i = v)$ over the component's attribute variables, and an associated reward $\mathbb{R}_i(x_i = v)$.

More precisely, the constraint automaton $\mathcal{C}_i$ for component i is described by a tuple $\langle \Pi_i, \mathbb{M}_i, \mathbb{T}_i, \mathbf{P}_{\Theta_i}, \mathbf{P}_{\mathbb{T}_i}, \mathbb{R}_i \rangle$, where:

1) Π_i is a set of variables for the component, where each var $\in \Pi_i$ ranges over a finite domain $\mathbb{D}(\text{var})$. Π_i is partitioned into a singleton set Π_i^m, containing the component's mode variable x_i, and a set Π_i^a of *attribute variables* a_i. Attributes include input variables, output variables, and any other variables needed to describe the modeled behavior of the component. $\mathbb{C}(\Pi_i)$ denotes the set of all finite-domain constraints over Π_i.
2) $\mathbb{M}_i : \mathbb{D}(x_i) \rightarrow \mathbb{C}(\Pi_i)$ associates with each mode assignment $x_i = v$ a finite domain constraint $\mathbb{M}_i(x_i = v) \in \mathbb{C}(\Pi_i)$. This constraint captures the component's behavior in a given mode.
3) $\mathbb{T}_i : \mathbb{D}(x_i) \times \mathbb{C}(\Pi_i) \rightarrow \mathbb{D}(x_i)$ associates with each mode assignment $x_i = v$ a set $\mathbb{T}_i(x_i = v)$ of transition functions. Each transition function $\tau_i(x_i = v) \in \mathbb{T}_i(x_i = v)$ specifies an assignment to x_i at time $t + 1$, conditioned on satisfaction of a constraint in $\mathbb{C}(\Pi_i)$ at time t. $\mathbb{T}_i(x_i = v)$ is the set $\{\tau_i^n(x_i = v), \tau_i^{f1}(x_i = v), \tau_i^{f2}(x_i = v), \ldots\}$, where the transition function representing nominal behavior is denoted $\tau_i^n(x_i = v)$, and the other transition functions represent fault behaviors.
4) $\mathbf{P}_{\Theta_i} : \mathbb{D}(x_i) \rightarrow \Re[0,1]$ denotes the probability that $x_i = v$ is the initial mode for component i.
5) $\mathbf{P}_{\mathbb{T}_i} : \mathbb{T}_i(x_i = v) \rightarrow \Re[0,1]$ denotes, for each mode variable assignment $x_i = v$, a probability distribution over the possible transition functions $\tau_i(x_i = v)$.
6) $\mathbb{R}_i : \mathbb{D}(x_i) \rightarrow \Re$ denotes the reward associated with mode variable assignment $x_i = v$.

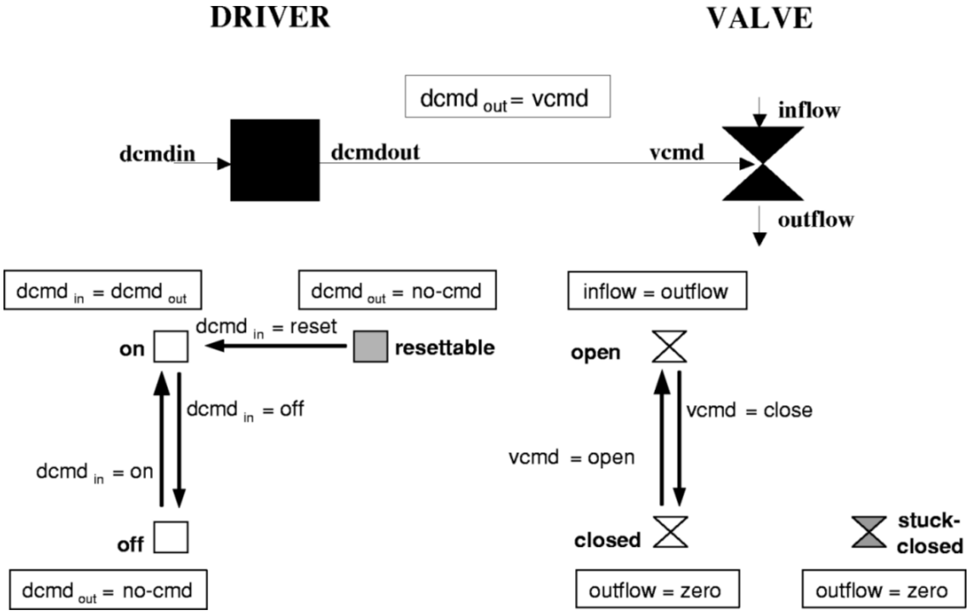


Fig. 16 Constraint automata for a driver component and a valve component. The driver has a single fault mode, "resettable," corresponding to a recoverable failure of the device. The valve is also modeled with a single fault mode, "stuck-closed." The probabilistic transitions from the nominal modes to these failure modes are omitted from the diagram for clarity.

It should be noted that these transitions correspond to a restricted form of the transitions defined for HCA. Transitions between successive modes are conditioned on constraints on variables Π_i. For each value v of component mode variable x_i, we represent each of its transition functions, $\tau_i(x_i = v)$, as a set of *transition* pairs (l, v'), where $v' \in \mathbb{D}(x_i)$, and l is a set of labels (or guards) of the form $\models c$ (denoted c) or $\not\models c$ (denoted $\overline{c}$), for some $c \in \mathbb{C}(\Pi_i)$. This corresponds to the traditional representation of transitions as labeled arcs in a graph, where v and v' are the source and target of an arc with label l. Unlike HCA transitions, we impose the restriction that, for each transition function, exactly one transition pair will be enabled for any mode assignment $x_i = v$. This means that a component can only be in a single mode at any given time.

B. Concurrent Constraint Automata

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

Formally, a CCA $\mathcal{P}$ is described by a tuple $\langle \mathcal{C}, \Pi, \mathbb{I} \rangle$, where the following conditions exist.

1) $\mathcal{C} = \{\mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_n\}$ denotes the finite set of constraint automata associated with the n components in the plant.
2) Π is a set of *plant variables*, where each var $\in \Pi$ ranges over a finite domain $\mathbb{D}(\text{var})$. $\mathbb{C}(\Pi)$ denotes the set of all finite-domain constraints over Π. Π is partitioned into sets of *mode* variables $\Pi^m = \bigcup_{i=1\ldots n} \Pi_i^m$, *control* variables $\Pi^c \subset \bigcup_{i=1\ldots n} \Pi_i^a$, *observable* variables $\Pi^o \subset \bigcup_{i=1\ldots n} \Pi_i^a$, and *dependent* variables $\Pi^d \subset \bigcup_{i=1\ldots n} \Pi_i^a$.
3) $\mathbb{I} \in \mathbb{C}(\Pi^c \cup \Pi^o \cup \Pi^d)$ is a conjunction of constraints providing the interconnections between the attributes of the plant 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. Finally, dependent variables represent interconnections between components. They are used to transmit the effects of control actions and observations throughout the plant model.

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

A CCA models the evolution of physical processes by enabling and disabling constraints in a *constraint store*. Enabled constraints in the store would include the set of $\mathbb{M}_i(x_i = v)$ constraints imposed by the current plant state and the $\mathbb{I}$ constraints associated with the CCA.

C. CCA Specification Example

Consider the valve and driver component models depicted in Fig. 16. The plant variables x_i associated with these components are Driver and Valve, with domains of {On, Off, Resettable} and {Open, Closed, Stuck-closed}, respectively. The driver's constraint automaton has attributes dcmd_{in} and dcmd_{out}. The valve's constraint automaton has attributes vcmd, inflow, and outflow.

The mode constraints $\mathbb{M}_i$ for the driver component are specified as follows:

$$\begin{aligned}\mathbb{M}_{\text{Driver}}(\text{Driver = On}) &= (\text{dcmd}_{\text{in}} = \text{dcmd}_{\text{out}})\\ \mathbb{M}_{\text{Driver}}(\text{Driver = Off}) &= (\text{dcmd}_{\text{out}} = \text{no-cmd})\\ \mathbb{M}_{\text{Driver}}(\text{Driver = Resettable}) &= (\text{dcmd}_{\text{out}} = \text{no-cmd}).\end{aligned}$$

For the driver component, two transition functions are specified, the nominal transition function τ^n_{Driver} and the failure transition function τ^f_{Driver}

$$\begin{aligned}\tau^n{}_{\text{Driver}}(\text{Driver = On}) &= \{(\text{dcmd}_{\text{in}} = \text{Off,Off}),\\ &\quad \times (\overline{\text{dcmd}_{\text{in}} = \text{Off}},\text{On})\}\\ \tau^n{}_{\text{Driver}}(\text{Driver = Off}) &= \{(\text{dcmd}_{\text{in}} = \text{On,On}),\\ &\quad \times (\overline{\text{dcmd}_{\text{in}} = \text{On}},\text{Off})\}\\ \tau^n{}_{\text{Driver}}(\text{Driver = Resettable}) &= \{(\text{dcmd}_{\text{in}} = \text{Reset,On}),\\ &\quad \times (\overline{\text{dcmd}_{\text{in}} = \text{Reset,}}\\ &\quad \text{Resettable})\}\\ \tau^f{}_{\text{Driver}}(\text{Driver = On}) &= \{(\text{True, Resettable})\}\\ \tau^f{}_{\text{Driver}}(\text{Driver = Off}) &= \{(\text{True, Resettable})\}\\ \tau^f{}_{\text{Driver}}(\text{Driver = Resettable}) &= \{(\text{True,Resettable})\}.\end{aligned}$$

Mode constraints and transition functions can be similarly expressed for the valve component.

The probabilities in $\mathbf{P}_{\mathbb{T}_{\text{Driver}}}$ associated with transition functions $\tau^n{}_{\text{Driver}}$ and $\tau^f{}_{\text{Driver}}$ are 0.99 and 0.01, respectively (in this case, the probabilities are independent of the current mode variable assignment). Modal costs and initial mode probabilities are not specified in this example.

For the CCA composed of the driver and valve constraint automata, we specify the set of control variables $\Pi^c = \{\text{dcmd}_{\text{in}}\}$, the set of observable variables $\Pi^o = \{\text{inflow}, \text{outflow}\}$, and the set of dependent variables $\Pi^d = \{\text{dcmd}_{\text{out}}, \text{vcmd}\}$.

The component interconnections are given by

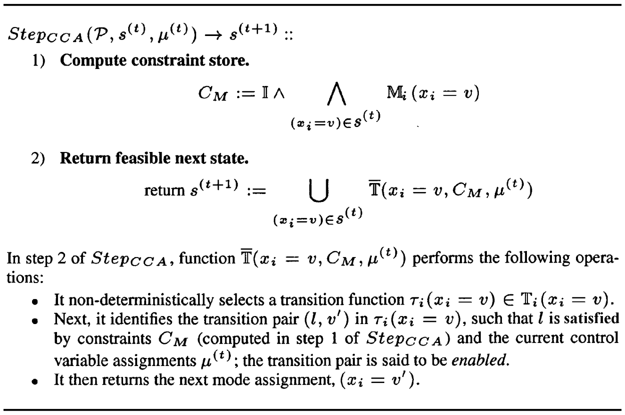
$$\mathbb{I} = (\text{dcmd}_{\text{out}} = \text{vcmd}).$$

D. Feasible Trajectories of Concurrent Constraint Automata

Next, consider the set of trajectories that are *feasible*, that is, trajectories in which each pair of sequential states is consistent with a CCA. We defer to the next section discussion of the computation of the most likely plant state, given a set of observations.

Given a sequence of control variable assignments $[\mu^{(0)}, \mu^{(1)}, \ldots]$, a *feasible trajectory* of a plant $\mathcal{P}$ is a sequence of plant states $[s^{(0)}, s^{(1)}, \ldots]$, such that: 1) $s^{(0)}$ is a valid initial plant state, i.e., $\mathbf{P}_{\Theta_i}(x_i = v) > 0$ for all assignments $(x_i = v) \in s^{(0)}$; and 2) for each $t \geq 0$, $s^{(t+1)} = \text{Step}_{\text{CCA}}(\mathcal{P}, s^{(t)}, \mu^{(t)})$.

$\text{Step}_{\text{CCA}} : \Sigma(\Pi^m) \times \Sigma(\Pi^c) \rightarrow \Sigma(\Pi^m)$ transitions the CCA of a plant $\mathcal{P}$, by nondeterministically executing a transition for each of its component automata, according to the algorithm in Fig. 17.



$Step_{CCA}(\mathcal{P}, s^{(t)}, \mu^{(t)}) \rightarrow s^{(t+1)}$ ::

1) **Compute constraint store.**

$$C_M := \mathbb{I} \wedge \bigwedge_{(x_i=v)\in s^{(t)}} \mathbb{M}_i\,(x_i = v)$$

2) **Return feasible next state.**

$$\text{return } s^{(t+1)} := \bigcup_{(x_i=v)\in s^{(t)}} \overline{\mathbb{T}}(x_i = v, C_M, \mu^{(t)})$$

In step 2 of $Step_{CCA}$, function $\overline{\mathbb{T}}(x_i = v, C_M, \mu^{(t)})$ performs the following operations:

- It non-deterministically selects a transition function $\tau_i(x_i = v) \in \mathbb{T}_i(x_i = v)$.
- Next, it identifies the transition pair (l, v') in $\tau_i(x_i = v)$, such that l is satisfied by constraints C_M (computed in step 1 of $Step_{CCA}$) and the current control variable assignments $\mu^{(t)}$; the transition pair is said to be *enabled*.
- It then returns the next mode assignment, $(x_i = v')$.

Fig. 17 Step_{CCA} algorithm.

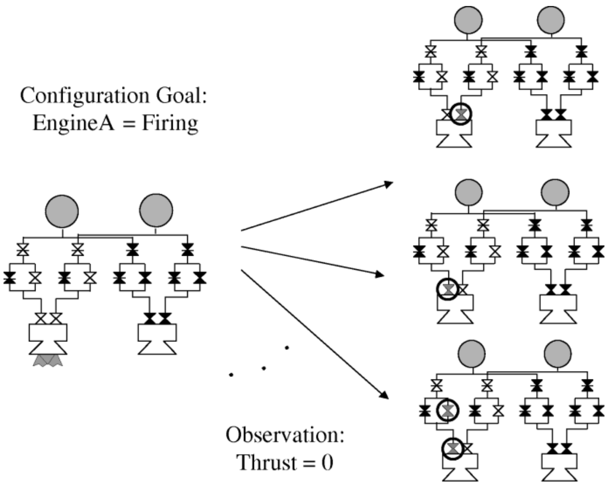


Fig. 18 One step of mode estimation for the orbital insertion example. Possible faulty valves are circled and closed valves are filled.

We use $\text{Step}_{\text{CCA}}{}^N$ to denote a variant of the step function that is restricted to only the nominal transition function τ^n_i. A trajectory that involves only nominal transitions τ^n_i is called a *nominal trajectory*.

E. Estimating Modes

Mode estimation incrementally tracks the set of plant trajectories that are consistent with the plant model, the sequence of observations, and the control actions. This is maintained as a set of consistent current states. For example, suppose the deductive controller is trying to maintain the configuration goal (EngineA = Firing), as shown on the left in Fig. 18. We assume that mode estimation starts with knowledge of the initial state, in which a set of open valves allows a flow of oxidizer and fuel into Engine A. In the next time instant, the sensors send back the observation that (Thrust = Zero). Mode estimation then considers the sets of possible component mode transitions that are consistent with this observation, given the initial state and plant CCA. It identifies a number of state transitions that are consistent with this observation, including that either of the inlet valves into Engine A has transitioned to stuck closed, or that any combination of valves along the fuel or oxidizer supply path are stuck closed, as depicted on the right in Fig. 18.

Unfortunately, the size of the set of possible current states is exponential in the number of components. Even for this simple example, this reaches about a trillion states. Furthermore, a large percentage of these states will not be ruled out by past observations. For example, given a system whose behavior looks correct based on current observations, it is always possible that *any combination of components*—from one to all—are broken in a novel manner, but have simply not manifested their symptoms yet.

This exponential set is infeasible to track in real time for most practical embedded systems. In addition, the set of consistent states is far too large to provide significant useful information. To perform useful actions, the deductive controller needs to be able to distinguish the small set of plausible current states from the large set of consistent but implausible states. To accomplish this, mode estimation computes the likelihood of the current estimated states, called a *belief state*. In addition, it computes the set of consistent trajectories and states in order of likelihood. This offers an any-time algorithm, by stopping when no additional computational resources are available to compute the less likely trajectories.

In Section III, we defined the plant model in terms of a POMDP, and discussed how mode estimation is framed as an instance of belief state update. In this section, we discuss the implementation of belief state for CCA. We then describe the simplest variant of mode estimation for CCA, based on a form of beam search.

1) Belief Update for CCA: To define belief update for CCA, we simply need to define $\mathbf{P}_{\mathbb{T}}$ and $\mathbf{P}_{\mathbb{O}}$. To be consistent with our earlier development, these definitions must constrain the probability of any inconsistent trajectory to be zero. To calculate the probabilistic transition function $\mathbf{P}_{\mathbb{T}}$ for state s_j of a plant CCA, we define a plant transition function to be composed of a set of component transition functions, one for each component mode $(x_i = v) \in s_j$. In addition, a CCA specifies the transition probability distribution for each component mode through $\mathbf{P}_{\mathbb{T}_i}(x_i = v)$. We make the key assumption that component transition probabilities are conditionally independent, given the current plant state s_j. This is analogous to the failure independence assumptions made by the General Diagnostic Engine (GDE) [19], Sherlock [20] and Livingstone [3], and is a reasonable assumption for most engineered systems. Hence, $\mathbf{P}_{\mathbb{T}}(s_j)$ is computed as a cross product of the probability distributions over the individual component transition functions

$$\mathbf{P}_{\mathbb{T}}(s_j) = \prod_{(x_i=v)\in s_j} \mathbf{P}_{\mathbb{T}_i}(x_i = v).$$

We calculate the observation function $\mathbf{P}_{\mathbb{O}}$ for s_j from the CCA, taking an approach similar to that of GDE [19]. Given the constraint store C_M for s_j, computed as the conjunction of $\mathbb{I}$ and $\mathbb{M}_i(x_i = v)$ for all $(x_i = v) \in s_j$, we test if each observation in o_k is entailed or refuted by the conjunction of C_M and s_j, giving $\mathbf{P}_{\mathbb{O}}$ probability 1 or 0, respectively. If no prediction is made, that is, 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 toward states that predict observations, over states that are merely consistent with the observations. These two definitions for $\mathbf{P}_{\mathbb{T}}$ and $\mathbf{P}_{\mathbb{O}}$ complete our belief update equations for CCA.

2) CCA Belief Update Under Tight Resource Constraints: The remaining step of our development is to implement mode estimation as an efficient form of limited belief update. Earlier in this section, we pointed out that the belief state contains an exponential number of states. In particular, the belief state is n^m, where m is the number of mode variables and n is the size of the domain of the mode variables. A real-world model of an embedded system, such as the model developed for NASA's DS-1 probe [3], involves roughly 80 mode variables and 3^{80} states. DS-1 allocated 10% of its 20–MHz processor to mode estimation and reconfiguration, and had an update rate for sensor values of once every second. These resource constraints allow only a small fraction of the state space to be explored explicitly in real time.

A standard approach, frequently employed for Hidden Markov Models, is to construct the Trellis diagram (see Fig. 6) over a finite, moving window, and to identify the most likely trajectory as a shortest path problem. In state estimation for CCA, the Trellis diagram forms a constraint graph with concurrent transitions, and the shortest path is found by framing a problem to the OpSat optimal constraint satisfaction engine [21].

For embedded systems with severe computational resource constraints, such as a spacecraft, we desire an approach that limits the amount of online search. This may be accomplished by tracking the most likely trajectories through the Trellis diagram, expanding only the highest probability transitions at each step, as time permits. This approach provides an any-time algorithm, as the most likely belief states are generated in best-first order.[4] From a practical standpoint, this approach has proven extremely robust on a range of space systems [3], including deep-space probes, reusable launch vehicles, and Martian habitats, and it far surpasses the monitoring capabilities of most current embedded systems.

3) Computing Likely Mode Estimates Using OpSat: Titan frames the enumeration of likely transition sets as an optimal constraint satisfaction problem (OCSP) and solves the problem using the OpSat algorithm, presented in detail in [21].

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 multiattribute 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

[4] The limitation is that a low probability trajectory may be pruned, which, after additional evidence is collected, could become highly likely.

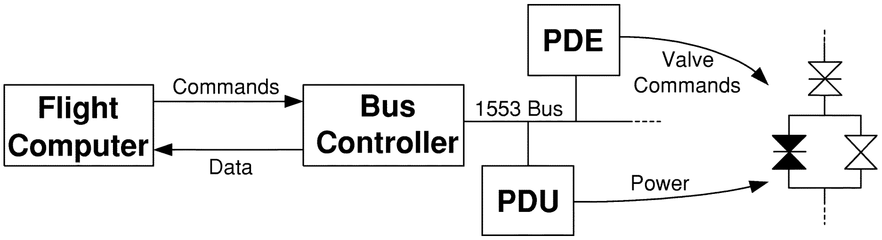


Fig. 19 A spacecraft has complex paths of interaction: commands issued by the flight computer pass through the bus controller, 1553 bus, and PDE on the way to the spacecraft main engine system.

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 leading candidates that are similarly inconsistent.

To view mode estimation as an OCSP, recall that each plant transition consists of a single transition for each of its component transition systems. Hence, we introduce a variable into $\mathbf{x}$ for each component in the plant whose values are the possible component transitions. Each plant transition corresponds to an assignment of values to variables in $\mathbf{x}$. $C(\mathbf{x})$ is used to encode the condition that the target state of a plant transition and its corresponding constraint store must be consistent with the observed values. The objective function, f, is just the prior probability of the plant transition. The resulting OCSP identifies the leading transitions at each state, allowing mode estimation to track the set of likely trajectories.

Note that the OCSP for a typical plant model is quite large. For example, the plant model developed for the DS-1 spacecraft consisted of roughly 80 mode variables, 3300 propositional variables, and 12 000 propositional clauses. The two reasoning methods we employ in OpSat to achieve fast response are conflict-directed A* [21] and incremental truth maintenance (ITMS) [22]. Consistency of a candidate solution is tested through systematic search and unit propagation. For our applications, the cost of performing unit propagation dominates. The ITMS offers an incremental approach to performing unit propagation that provides an order of magnitude performance improvement over traditional truth maintenance systems. If the ITMS finds a candidate to be inconsistent, it returns a conflict associated with the inconsistency. Conflict-directed A* then uses the conflicts returned by the ITMS to prune inconsistent subspaces during best-first search over decision variables $\mathbf{x}$. Conflict-directed A* dramatically reduces the number of candidate states tested for consistency during mode estimation; for most applications, the number is typically less than a dozen states.

F. Reconfiguring Modes

Mode reconfiguration takes as input a configuration goal and returns a series of commands that progress the plant toward a maximum-reward state that achieves the configuration goal. Mode reconfiguration accomplishes this through two capabilities, the *goal interpreter* and *reactive planner*. These capabilities operate in tight coordination with mode estimation, described in the preceding section. The goal interpreter uses the plant model and the most likely current state, provided by mode estimation, to determine a reachable goal state that achieves the configuration goal, while maximizing reward. The reactive planner takes a goal state and a current mode estimate, and generates a command sequence that moves the plant to this goal state. The reactive planner generates and executes this sequence one command at a time, using mode estimation to confirm the effects of each command. For example, in our orbital insertion example, given a configuration goal of (EngineA = Firing), goal interpretation selects a set of valves to open, in order to establish a flow of fuel into the engine. Reactive planning then sends commands to control units, drivers, and valves to achieve this goal state.

Having identified which valves to open and close, one might imagine that achieving the configuration is a simple matter of calling a set of open-valve and close-valve routines. This is in fact how Titan's predecessor, Livingstone [3], performed mode reconfiguration. However, much of the complexity of mode reconfiguration is involved in correctly commanding each component to its intended mode, through lengthy communication paths from the control processor.

For example, Fig. 19 shows a typical set of communication paths from the flight computer to part of the spacecraft main engine system. The flight computer sends commands to a bus controller, which broadcasts these commands over a 1553 data bus. These commands are received by a bank of device drivers, such as the propulsion drive electronics (PDE). Finally, the device driver for the appropriate device translates the commands to analog signals that actuate the device.

The following is an example of a scenario that a robust close-valve routine should be able to handle:

> The close-valve routine should first ascertain if the valve is open or closed, by polling its sensors. If it is open, it needs to broadcast a "close" command. However, first it needs to determine if the driver for the valve is on, again by polling its sensors, and if not, it should send an "on" command. Now suppose that shortly after the driver is turned on, it transitions to a resettable failure mode. The valve driver needs to catch this failure, and then before sending the "close" command, it should issue a "reset" command. Once the valve has closed, the close-valve routine should power off the valve driver, to save power. However, before doing so, it should change the state of any other valve that is controlled by that driver; otherwise, the driver will need to be immediately turned on again, wasting time and power.

There are several properties that make mode reconfiguration 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 to that device, by changing the modes of other devices. 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. Finally, corrective action should be performed as soon as a failure is detected, in order to mitigate further damage.

The challenge for mode reconfiguration is twofold. First, controlling a plant, described by CCA, generalizes the classic artificial intelligence (AI) planning problem [23], which is already NP-hard, to one of handling indirect effects. Second, to be practical for embedded systems, the deductive controller's response time should be driven toward a small constant. Accomplishing this requires eliminating as many instances of online search as possible.

1) Robustness Through Reversibility: A key decision underlying our model-based executive is the focus on the most likely trajectory generated by mode estimation. Livingstone [3] took a more conservative strategy of considering a single control sequence that covers the complete set of likely states. The difficulty with this approach is that the different estimated states will frequently require different control sequences.

Utility theory can be used to select, among different control sequences, one that maximizes the expected likelihood of success [24]. However, the cost of generating multiple states and control sequences works against our goal of building a fast reactive executive. Achieving this goal is essential if model-based programming languages are to be employed within everyday embedded systems, where procedural programming is the norm.

Instead, we adopt a greedy approach of assuming the most likely estimated state is the correct state. This greedy approach 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 model-based planner 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 forever cuts off access to parts of the propulsion system. Such actions should be taken after a human operator or high-level planning system explicitly reasons through the consequences of the actions over a future time horizon. Hence, fundamental to the reliability of our approach is the requirement that MR will only generate reversible control actions, unless the purpose of the action is to repair failures.[5]

2) Goal Interpretation (GI): GI is closely related to mode estimation. While mode estimation searches for likely modes that are consistent with the observations, GI searches for modes that entail the configuration goal while maximizing reward. GI generates a maximum-reward plant state $s_g^{(t)}$ that satisfies the current goal configuration $g^{(t)}$ and that is reachable from the current most likely state $\hat{s}^{(t)}$, using nominal transitions whose effects are *reversible*. We define reversible transitions $\mathbb{T}^R$ as transitions for which the source state may be returned to from the target state by a sequence of nominal control actions. We use $\mathbb{T}^R(\hat{s}^{(t)})$ to denote the set of all states that are reachable from $\hat{s}^{(t)}$ through repeated application of reversible transitions. We call these states "reversibly reachable" from $\hat{s}^{(t)}$.

```
$ComputeGoalStates(\mathcal{P}, \hat{s}^{(t)}, g^{(t)}) \rightarrow S ::$
    1) Compute reversibly reachable target states.
        $RR := \prod_{(x_i=v) \in \hat{s}^{(t)}} \mathbb{T}_i^R(x_i = v)$
    2) Computes enabled constraints for each target.
        $EC := \left\{ \langle s, C_M \rangle \mid s \in RR, C_M = \mathbb{I} \wedge \bigwedge_{(x_i=v) \in s} \mathbb{M}_i(x_i = v) \right\}$
    3) Return consistent targets that entail goal.
        return $S := \left\{ s \mid \langle s, C_M \rangle \in EC, s \wedge C_M \text{ is consistent}, s \wedge C_M \models g^{(t)} \right\}$
```

Fig. 20 *ComputeGoalStates* algorithm.

A priori, computing $\mathbb{T}^R(\hat{s}^{(t)})$ seems like an expensive task. It could be computed by generating all trajectories of reversible transitions originating at $\hat{s}^{(t)}$, for example, by employing a symbolic reachability analysis similar to those used by symbolic model checkers. However, $\mathbb{T}^R(\hat{s}^{(t)})$ has the key property that it may be computed as the cross product of reachable target mode assignments, $\mathbb{T}_i^R(x_i = v)$, of the reversible transitions for individual components x_i

$$\mathbb{T}^R(\hat{s}^{(t)}) = \prod_{(x_i=v) \in \hat{s}^{(t)}} \mathbb{T}_i^R(x_i = v).$$

This property follows from subtle arguments related to reversibility of mode assignments and transitions [4]. The goal interpretation function, ComputeGoalStates : $\Sigma(\Pi^m) \times \Sigma(\Pi) \rightarrow 2^{\Sigma(\Pi^m)}$, employs this property to compute the set of all reversibly reachable target states S that achieve configuration goal $g^{(t)}$, given a plant model $\mathcal{P}$ with estimated current state $\hat{s}^{(t)}$. This algorithm is presented in Fig. 20.

Given reversibly reachable states $\mathbb{T}_i^R(x_i = v)$ for each component mode variable x_i, casting GI as an OCSP $\langle \mathbf{x}, f, C \rangle$ is straightforward. There is a decision variable in $\mathbf{x}$ for each x_i, with domain $\mathbb{T}_i^R(x_i = v)$, representing the reversibly reachable modes of component x_i. $C(\mathbf{x})$ 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 $g^{(t)}$. f is the reward, $\sum_{(x_i=v) \in s} \mathbb{R}_i(x_i = v)$, of being in a state s. An example of a reward metric is power consumption. Once again, Titan solves this OCSP using OpSat [21].

Fig. 21 shows an example of a conflict-directed search sequence, which OpSat performs during GI. Suppose we have the configuration goal (Spacecraft = Thrusting), which

[5] While repairing a failure is irreversible, it is important to allow a reactive executive to repair failures, in order to increase functionality.

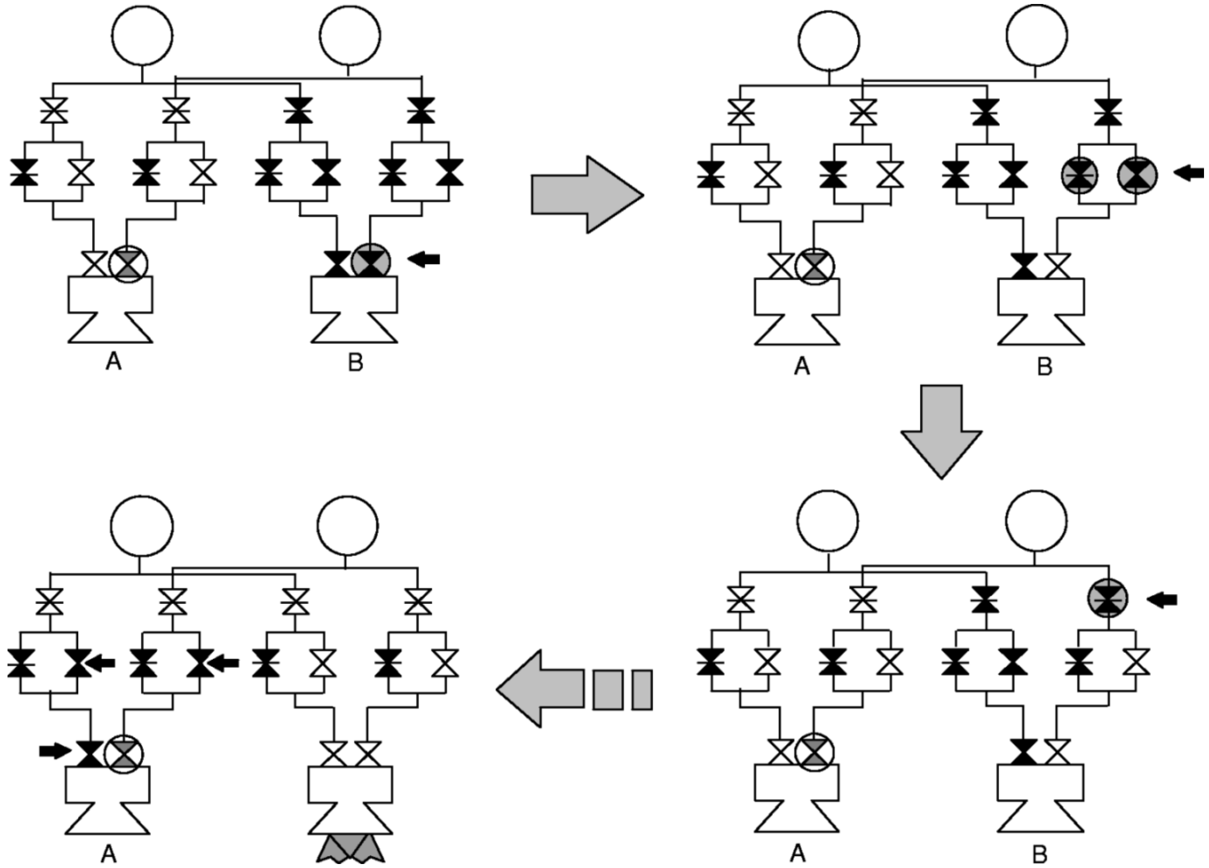


Fig. 21 Snapshots demonstrating how the goal interpreter searches for a reachable state that achieves the configuration goal of *(Spacecraft = Thrusting)* with maximum reward. Conflicts used to direct the search are highlighted by the small black arrows. In the last step, GI identifies that the open valves on the disabled Engine A should be closed, to minimize loss of propellant.

is achievable by firing either engine. Furthermore, suppose that the right flow inlet valve to Engine A has just stuck closed. This is the configuration shown in the upper left of Fig. 21, with the stuck valve circled and closed valves filled in. Starting with this configuration as a candidate goal state, GI deduces that it does not entail (Spacecraft = Thrusting). Furthermore, it extracts the conflict that thrust cannot be achieved unless the right inlet valve on either engine is opened. The right inlet valve on Engine A is stuck closed and cannot be made open. To resolve the conflict, OpSat creates a new candidate state, which has the right inlet valve to Engine B open, as shown in the upper right of Fig. 21. OpSat tests this candidate and proves that it also does not entail thrust. In addition, it extracts the conflict that thrust will not be achieved as long as one of the two highlighted valves feeding Engine B is closed, or the stuck inlet valve to Engine A remains closed. OpSat generates the next candidate [Fig. 21 (lower right)], by opening one of the two highlighted valves leading into Engine B (the one with higher reward associated with its open mode). OpSat continues this process, and finds the maximum-reward goal state that achieves thrust [Fig. 21 (lower left)], after testing less than ten candidates. This is an exceptionally small number of tested candidates, given that the state space for this example contains roughly 3^{20} possible target states.

3) Reactive Planning: Given a goal state supplied by GI, the reactive planner (RP) is responsible for achieving this goal state. Titan's model-based RP is called Burton and was first introduced in [4]. RP takes as input the current most likely state and the goal state selected by GI, and generates the first control action that moves the plant toward the goal state. More precisely, RP takes as input a CCA model of a plant $\mathcal{P}$, a most likely current state $\hat{s}^{(t)}$ (from ME), and a maximum-reward goal state $s_g^{(t)}$ (from GI) that satisfies goal $g^{(t)}$. RP generates a control action $\mu^{(t)}$ such that $\hat{s}^{(t+1)} = \mathrm{Step}_{\mathrm{CCA}}{}^{N}(\mathcal{P}, \hat{s}^{(t)}, \mu^{(t)})$ is the goal state $s_g^{(t)}$, or $\langle \hat{s}^{(t)}, \hat{s}^{(t+1)} \rangle$ is the prefix of a simple nominal trajectory that ends in $s_g^{(t)}$.

Titan's model-based reactive planning problem closely relates to the classic AI planning problem [23].[6] The key difference is that a classical planner invokes operators that *directly* modify state. Titan's model-based RP, on the other hand, exerts control by establishing values for control variables, which interact *indirectly* with internal plant state variables, through cotemporal, physical interactions, represented

[6] The classic AI planning problem involves generating a set of discrete actions that move the system from an initial state to a goal state, where each action is an instance of a STRIPS operator [25]. A STRIPS operator is an atomic action that maps states to states. It is specified by a set of preconditions and a set of effects, which enumerate all changes in variable values.

in the CCA. This extension adds substantial expressivity to the planner.

The challenge is to preserve this expressivity, while providing a planner that is reactive. Titan's RP meets five desiderata. First, it only generates *nondestructive actions*, that is, an action will never undo the effects of previous actions that are still needed to achieve top-level goals. Second, RP will not propose actions that lead to *deadend plans*, that is, it will not propose an action to achieve a subgoal when one of the sibling subgoals is unachievable. Third, the reactive planner is *complete*, that is, when given a solvable planning problem, it will generate a valid plan. Note, however, that RP may not generate the shortest length plan. Fourth, RP ensures *progress* to a goal, except when execution anomalies interfere, that is, the nominal trajectory generated by RP for a fixed goal is loop-free. Fifth, RP operates at *reactive time scales*—its average run-time complexity is constant. This speed is essential to providing a model-based executive with response times comparable to traditional procedural executives [26], [27].

Classical planners search through the space of possible plans, while using mechanisms, such as *threat detection*, to determine if one action is in danger of prematurely removing an effect intended for a subsequent action [23]. Titan's RP avoids run-time search, requires no algorithms for threat detection, and expends no effort determining future actions or planning for subgoals that are not supported by the first action. The RP accomplishes this speedup 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 (that is, one at a time), using divide and conquer to achieve an exponential speedup.

A complete development of RP is beyond the scope of this paper. The design of the Burton reactive planner has been previously presented in [4]. In the remainder of this section, we highlight three key elements of RP: compilation of concurrent constraint automata into concurrent automata without constraints, generation of concurrent policies from the compiled concurrent automata, and online execution of the concurrent policies.

Compiling Away Indirect Interactions: Indirect control and effects substantially complicate planning. We solve the problem of indirect control and effects at compile-time, by mapping the plant CCA to an equivalent automaton that only has *direct* effects. In particular, the transition labels of a CCA contain conditions involving dependent variables that are not directly controllable, but depend on control variables that are related through the mode constraints $\mathbb{M}_i$ and the interconnections $\mathbb{I}$. In the slightly modified driver/valve example of Fig. 22, the valve transitions are conditioned on assignments to the dependent variable $\mathrm{vcmd}_{\mathrm{in}}$, which is linked to control variable $\mathrm{dcmd}_{\mathrm{in}}$ by the interconnection constraints and the mode constraints of the driver [see Fig. 22(a)].

To eliminate indirect control, we eliminate dependent variables from each transition label, by constructing an equivalent label in terms of control and mode variables alone. For example, we replace the valve's $\{\mathrm{vcmd}_{\mathrm{in}} = \mathrm{open}\}$ transition label with the pair of conditions {driver = on, $\mathrm{dcmd}_{\mathrm{in}} =$ open} [see Fig. 22(b)]. Once the dependent variables have been eliminated from the labels, the mode constraints $\mathbb{M}_i$ and interconnections $\mathbb{I}$ are no longer needed, and hence are removed, as shown in Fig. 22(b). The result is a set of concurrent automata without constraints, that are conditioned on control variable assignments (called *control conditions*) and mode variable assignments (called *state conditions*).

Compilation is performed by a restricted form of *prime implicant generation* that is described in [4]. While prime implicant generation is NP-hard, in practice even sizable sets of implicants can be generated very fast. We use OpSat [21] to perform a form of abductive best-first search for prime implicants in the plant model. This allows us to generate implicants from a spacecraft model consisting of over 12 000 clauses in about 40 s on a Sparc 20. We compute these implicants at compile-time, thus avoiding the impact of these computations on run-time performance.

Generating Concurrent Policies: Once the constraints have been compiled out of the concurrent automata, we use these automata to generate control codes for each component, which form a set of concurrent *feasible policies*. For each possible current mode assignment $x_i = v_c$ and goal mode assignment $x_i = v_f$, a feasible policy π_x maps pairs $\langle v_c, v_f \rangle$ to the ordered list of state conditions and control conditions that must be achieved in order to transition from v_c to v_f (see Fig. 23). For example, given the goal state of (valve = open), and current state (valve = closed), the valve policy says that the plan must first achieve the goal state (driver = on).

To generate such policies, RP imposes the following requirements on the compiled plant model.

1) Each control variable has an idling assignment, and no idling assignment appears in any transition. The label of every transition includes a (nonidling) control assignment or a mode assignment subgoal.
2) No set of control assignments of one transition is a proper subset of the control assignments of a different transition.
3) The causal graph for the compiled concurrent automaton must be *acyclic*. A *causal graph* $\mathbb{G}$ for a compiled plant model is a directed graph whose vertices are the state variables of the plant. $\mathbb{G}$ contains an edge from variable x_i to x_j, if x_i appears in the label of any of x_j's transition pairs (l, v). In the compiled model for the driver/valve example, the state variable *driver* appears in two of the transitions for the *valve* component, so we draw an edge from *driver* to *valve* in the causal graph (see Fig. 24).

The basic idea behind these policies is that we wish to solve a conjunction of goal assignments by achieving conjuncts one at a time—that is, *serially*—and by ordering goal achievement by working "upstream" along the acyclic causal graph, from outputs to control variables. This upstream ordering corresponds to a topological numbering. We establish a *topological order* among component mode

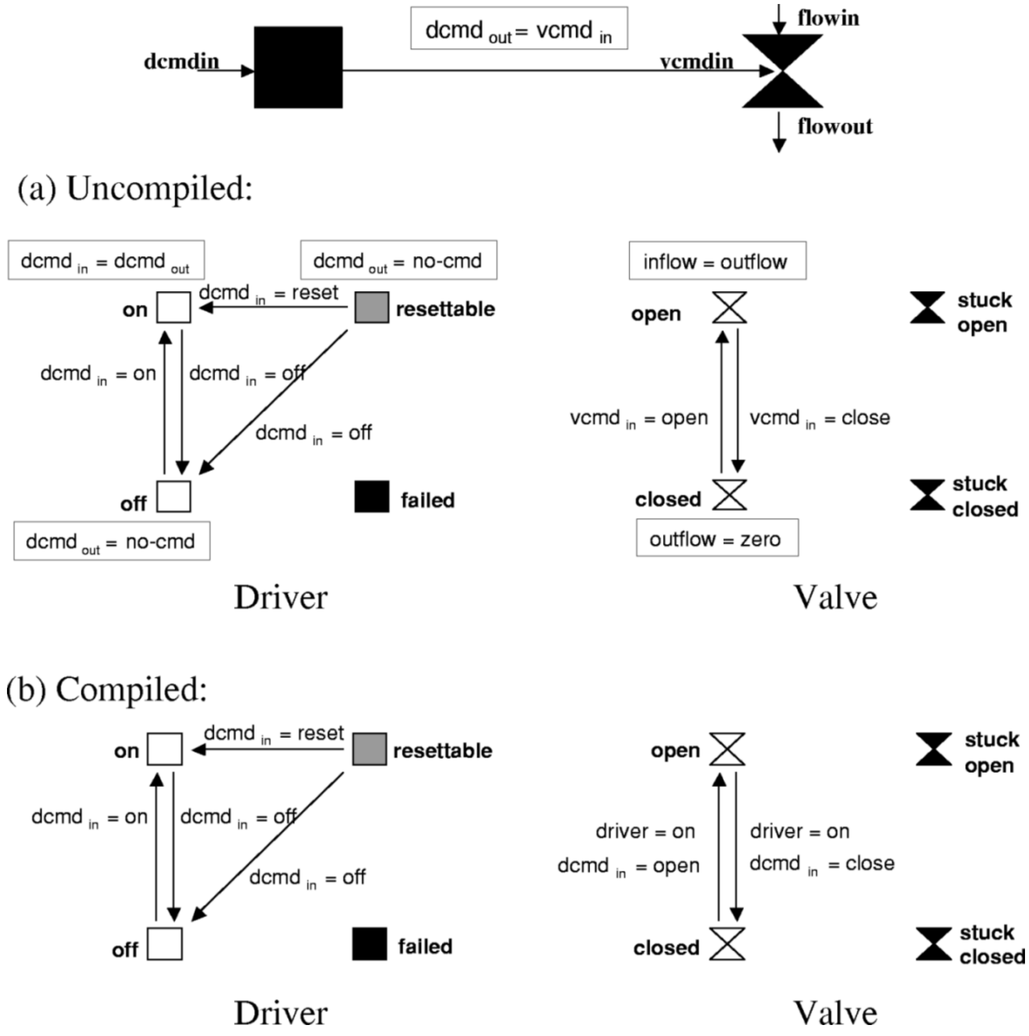


Fig. 22 Compilation example for a driver/valve pair. The uncompiled concurrent constraint automata are shown in (a) and the compiled concurrent automata are shown in (b).

driver	Target	
Current	*on*	*off*
on	**Idle**	$dcmd_{in} = off$
off	$dcmd_{in} = on$	**Idle**
resettable	$dcmd_{in} = reset$	$dcmd_{in} = off$
failed	**Failure**	**Failure**

valve	Target	
Current	*open*	*closed*
open	**Idle**	$driver = on$, $dcmd_{in} = close$
closed	$driver = on$, $dcmd_{in} = open$	**Idle**
stuck open	**Failure**	**Failure**
stuck closed	**Failure**	**Failure**

Fig. 23 Policies for the driver (top) and valve (bottom).

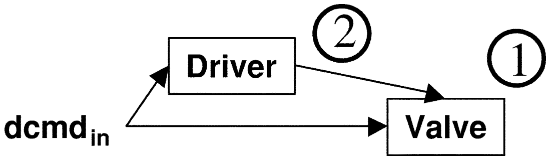


Fig. 24 A causal graph for the driver/valve pair. This causal graph is acyclic. Topological numbers are circled.

variables by performing a depth-first search of the causal graph at compile-time, and numbering variables on the way out. For our example, this process determines the topological order as indicated in Fig. 24. By encoding serialized goals in the policies, we eliminate the potential for conflicting subgoal interactions, thus guaranteeing that RP generates nondestructive action sequences. Proof of this fact is provided in [4]. Achieving subgoals serially also ensures that RP always makes progress toward the goal (Desiderata 4).

For a mode variable x with domain size n, x's policy π_x is of size n^2, and the policy is computed in $O(n * e)$ time, where e is the maximum number of transitions in a single component constraint automaton. Entries for each v_c are determined by computing a spanning tree directed toward v_f that connects all reversibly reachable modes, by reversible transitions.

RP's concurrent policies are analogous to traditional optimal policies in control theory and universal plans in AI. An important difference is that Titan's RP constructs a set of concurrent policies, rather than a single policy for the complete

$ComputeNextAction(\hat{s}^{(t)}, s_g^{(t)}, \pi, \textbf{top?}) \rightarrow \mu^{(t)}$ ::

1) **Goals are solvable?** When **top?** $=$ **True**, unless each goal assignment $g \in s_g^{(t)}$ is reversibly reachable, return **Failure.**
2) **Select unachieved goal.** Find the unachieved goal assignment with the lowest topological number. Goal $(x = v_f) \in s_g^{(t)}$ is unachieved if it differs from x's current assignment $(x = v_c) \in \hat{s}^{(t)}$. If all goal assignments are achieved, return **Success.**
3) **Select next transition.**: $\langle SC, CC \rangle = \pi_x(v_c, v_f)$ Let SC and CC be the state and control conditions of the first transition in the trajectory from v_c to v_f (along reversible transitions).
4) **Return control action.** Unless $SC = \{\}$, *Control* = *ComputeNextAction*($\hat{s}^{(t)}$, SC, π, *False*). If $SC = \{\}$ or *Control* $=$ **Success**, then state conditions SC are already satisfied, return CC to effect transition. Otherwise, *Control* contains control assignments that will make progress on achieving SC. Return *Control.*

Fig. 25 ComputeNextAction algorithm.

state space. A traditional policy grows exponentially in the number of state variables, and is infeasible for models such as a spacecraft, which contain more than 80 state variables and 3^{80} states. In contrast, Titan's concurrent policies grow only linearly in the number of state variables, and have the additional feature of preserving modularity.

Online Policy Execution: Once the concurrent policies have been generated at compile-time, planning is performed using the online reactive planning function ComputeNextAction, presented in Fig. 25. ComputeNextAction takes as input the current state estimate $\hat{s}^{(t)}$, a goal state $s_g^{(t)}$, compiled concurrent policies π and the flag **top?**, used to indicate a top-level call. ComputeNextAction returns either a next control action, **Failure** if no plan exists, or **Success** if the current state is the goal state. The assignments in $s_g^{(t)}$ are sorted by topological number, providing an upstream order in which the goals may be solved serially. A control action is a partial set of control assignments. All control variables not mentioned are assigned their idle value.

Step 1 of ComputeNextAction tests whether or not a conjunction of top-level goals can be achieved. This operation can be performed very quickly, because the set of reversibly reachable modes, $\mathbb{T}_i^R(x_i = v)$, has already been computed during the offline CCA compilation process. Since the RP only introduces subgoal assignments that are guaranteed to be achievable, the test is needed at the top level only. Step 2 works upstream along the causal graph of variables, selecting the next unsatisfied (sub)goal assignment to be achieved. Step 3 takes the first step toward achieving the selected (sub)goal. Given a current assignment $x = v_c$, a goal assignment $x = v_f$ is achieved by traversing a path along transitions of x from v_c to v_f. Step 3 identifies the first transition along this path from v_c to v_f. Step 4 recursively calls ComputeNextAction on the subgoals comprising the state conditions of this first transition, which results in the RP moving further upstream. If one or more state conditions of the transition is unsatisfied, then a next control action is computed in Step 4, and returned. If all state conditions are satisfied, then the transition is ready to be executed, and Step 4 returns the transition's control conditions as the next control action.

For example, suppose mode estimation identifies the current state as $\{$driver = off, valve = open$\}$, and goal interpretation provides the goal $\{$valve = closed, driver = off$\}$, which is determined to be reversibly reachable (Step 1). Step 2 selects the first goal assignment in the topological order, $\{$valve = closed$\}$. Step 3 looks up open $\rightarrow$ closed in the valve policy π_{valve} (see Fig. 23), and gets SC $= \{$driver = on$\}$ and CC $= \{\text{dcmd}_{\text{in}} = \text{close}\}$. The first is an unsatisfied state condition, so Step 4 recursively calls ComputeNextAction with (driver = on) as the goal. Looking up off $\rightarrow$ on in the driver policy π_{driver}, we get the control condition $\{\text{dcmd}_{\text{in}} = \text{on}\}$. This control assignment is returned by ComputeNextAction and is immediately invoked.

For a fixed goal and no intervening failures, RP generates successive control actions as a depth-first traversal through the policy tables. This traversal maps out a subgoal tree, and RP maintains an index into this tree during successive calls. To generate a sequence, RP traverses each tree edge exactly twice, and generates one control action per vertex. Since the number of edges in a tree is bounded by the number of vertices, the *average case complexity of generating each control action is roughly constant.*[7] Thus, Desiderata 5—reactivity—is achieved.

Failure States and Repair: A key function of RP is to handle failure. To accomplish this, we incorporate repair actions. The occurrence of failures are outside the reactive planner's control, since there are no nominal transitions that lead to failure. Hence, a repair sequence is irreversible, albeit essential, and so is not covered by the development thus far. Titan's RP permits repair sequences, which are selected so as to minimize the number of irreversible steps taken. RP never uses a failure to achieve a goal assignment if the failure is repairable. However, if it is not repairable, then RP is allowed to exploit the component's faulty state. For example, suppose a valve is needed to be open, and it is permanently stuck-open. Since stuck-open is irreparable but has the desired effect, RP exploits the failure mode. The algorithm used by RP to invoke appropriate repair actions is presented in [4].

Returning to our driver/valve example, suppose mode estimation reports that the driver has just turned on as a result of our last call to ComputeNextAction. The current and goal assignments of the valve are unchanged, hence the entry $\langle$sc,cc$\rangle$ $=$ (driver = on, $\text{dcmd}_{\text{in}} = \text{close}$) is revisited. However, the first conjunct, (driver = on), is now satisfied. Since no state conditions remain, RP issues ($\text{dcmd}_{\text{in}} = \text{close}$) as the next action. Next, assume mode estimation reports that a partial failure occurred after the last command was issued, such that the current state is (driver = resettable, valve = closed). In its next call to ComputeNextAction, RP determines that the first unachieved goal assignment is now (driver = off). Looking up resettable $\rightarrow$ off in the driver policy π_{driver} (see Fig. 23) results in the repair action ($\text{dcmd}_{\text{in}} = \text{off}$).

[7] The caveat is that at each node all of the assignments of a label's transition may be visited, in order to find the next one that needs to be achieved. A label typically contains only a couple of assignments; thus, its size can generally be bounded by a small constant, but in the worst case this is on the order of the number of concurrent automata.

VII. Discussion

The RMPL compiler is written in C++. It generates HCA from RMPL control programs. The compiler supports all RMPL primitive combinators, and a variety of derived combinators. Plant models are currently written in MPL, the modeling language used for Livingstone. Compilation of the models into CCA is performed using Livingstone's Lisp-based compiler. The Titan model-based executive is written in C++, and builds on the OpSat system [21]. Titan is being demonstrated on mission scenarios for NASA's MESSENGER mission, and will be flight demonstrated on the MIT SPHERES spacecraft *inside* the International Space Station. Titan is a superset of the Livingstone system [3], which was demonstrated in flight on the DS-1 mission, and on a range of testbeds for NASA, the U.S. Navy and industry, including a space interferometer [9], a Mars *in situ* propellant production plant [10], and a Mars rover prototype [28].

Turning to related work, the model-based programming paradigm synthesizes ideas underlying synchronous programming, concurrent constraint programming, traditional robotic execution, and Markov models. Synchronous programming languages [11], [12] 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. One major goal of synchronous programming is to provide "executable specifications," that is, to eliminate the gap between the specifications about which we prove properties, and the programs that are supposed to implement these specifications. We carry this one step further by performing our reasoning on executable programs directly, in real time.

Model-based programming and concurrent constraint programming share common underlying principles, including the notion of computation as deduction over systems of partial information [29]. 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 RAPS [26] and ESL [27]. One key difference is that RMPL's constructs fully cover synchronous programming, hence moving toward 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 toward a unification with model-based autonomy.

We are pursuing a number of extensions to the model-based programming language and executive presented here. For example, in [7] we extend the model-based programming paradigm to include fast temporal planning. In [30], we extend the mode estimation capability of the executive to handle hybrid concurrent constraint automata, whose constraints are linear ordinary differential equations. Finally, we are extending RMPL to coordinate heterogeneous cooperative systems, such as fleets of unmanned air vehicles and Mars explorers [31], by unifying high-level mission planning with agile path planning.

Appendix
Derived RMPL Combinators

The following useful derived combinators can be constructed from the primitive constructs presented in Section IV-B2:

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 the 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 the variables of the physical plant. This expression is equivalent to {**always if** $(a \wedge c)$ **thennext** A, **do always** a **watching** c}, where a is a nonphysical state assertion introduced for guarding against starting A after the first instant in which c is entailed. a is trivially achieved when asserted.

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 the variables of the physical plant. This expression is equivalent to {**always if** c **thennext** A}.

Acknowledgment

The authors would like to thank A. Barrett, L. Fesq, B. Laddaga, M. Pekala, R. Ragno, H. Shrobe, G. Sullivan, J. Van Eepoel, D. Watson, A. Wehowsky, and D. Weld. They extend a special acknowledgment to V. Gupta for his contributions to the development of model-based execution.

References

[1] T. Young *et al.*. (2000) Report of the Mars Program Independent Assessment Team. NASA, Washington, DC. [Online] Available: http://www.nasa.gov/newsinfo/marsreports.html

[2] J. Casani *et al.*, "Report on the Loss of the Mars Polar Lander and Deep Space 2 Missions," NASA Jet Propulsion Laboratory, Pasadena, CA, JPL D-18 709, 2000.

[3] B. C. Williams and P. Nayak, "A model-based approach to reactive self-configuring systems," in *Proc. 13th Nat. Conf. Artif. Intell. (AAAI-96)*, vol. 2, 1996, pp. 971–978.

[4] ——, "A reactive planner for a model-based executive," in *Proc. Int. Joint Conf. Artif. Intell. (IJCAI-97)*, vol. 2, 1997, pp. 1178–1185.

[5] M. Ingham, R. Ragno, and B. C. Williams, "A reactive model-based programming language for robotic space explorers," presented at the Int. Symp. Artif. Intell., Robotics, Automation Space (ISAIRAS-01), Montreal, QB, Canada, 2001.

[6] S. Chung, J. V. Eepoel, and B. C. Williams, “Improving model-based mode estimation through offline compilation,” presented at the Int. Symp. Artif. Intell., Robotics, Automation Space (ISAIRAS-01), Montreal, QB, Canada, 2001.

[7] P. Kim, B. C. Williams, and M. Abramson, “Executing reactive, model-based programs through graph-based temporal planning,” in *Proc. Int. Joint Conf. Artif. Intell. (IJCAI-01)*, vol. 1, 2001, pp. 487–493.

[8] D. Bernard *et al.*, “Design of the Remote Agent Experiment for spacecraft autonomy,” presented at the IEEE Aerosp. Conf., Aspen, CO, 1999.

[9] M. Ingham *et al.*, “Autonomous sequencing and model-based fault protection for space interferometry,” presented at the Int. Symp. Artif. Intell., Robotics, Automation Space (ISAIRAS-01), Montreal, QB, 2001.

[10] C. Goodrich and J. Kurien, “Continuous measurements and quantitative constraints—challenge problems for discrete modeling techniques,” presented at the Int. Symp. Artif. Intell., Robotics, Automation Space (ISAIRAS-01), Montreal, QB, Canada, 2001.

[11] G. Berry and G. Gonthier, “The synchronous programming language ESTEREL: Design, semantics, implementation,” *Sci. Comput. Program.*, vol. 19, no. 2, pp. 87–152, 1992.

[12] D. Harel, “Statecharts: A visual formulation for complex systems,” *Sci. Comput. Program.*, vol. 8, no. 3, pp. 231–274, 1987.

[13] R. Davis, “Diagnostic reasoning based on structure and behavior,” *Artif. Intell.*, vol. 24, pp. 347–410, 1984.

[14] L. Fesq *et al.*, “Model-based autonomy for the next generation of robotic spacecraft,” in *Proc. 53rd Int. Astronautical Cong. Int. Astronautical Federation (IAC-02)*, 2002, Paper IAC-02-U.5.04.

[15] V. Saraswat, “The category of constraint systems is cartesian-closed,” presented at the 7th IEEE Symp. Logic Comput. Sci., Santa Cruz, CA, 1992.

[16] V. Saraswat, R. Jagadeesan, and V. Gupta, “Foundations of Timed Default Concurrent Constraint programming,” presented at the 9th IEEE Symp. Logic Comput. Sci., Paris, France, 1994.

[17] B. C. Williams, S. Chung, and V. Gupta, “Mode estimation of model-based programs: Monitoring systems with complex behavior,” in *Proc. Int. Joint Conf. Artif. Intell. (IJCAI-01)* , vol. 1, 2001, pp. 579–590.

[18] N. Halbwachs, *Synchronous Programming of Reactive Systems*, ser. Series in Engineering and Computer Science. Norwell, MA: Kluwer, 1993.

[19] J. de Kleer and B. C. Williams, “Diagnosing multiple faults,” *Artif. Intell.*, vol. 32, no. 1, pp. 97–130, 1987.

[20] ——, “Diagnosis with behavioral modes,” in *Proc. Int. Joint Conf. Artif. Intell. (IJCAI-89)*, 1989, pp. 1324–1330.

[21] B. C. Williams and R. J. Ragno, “Conflict-directed A* and its role in model-based embedded systems,” J. Discrete Appl. Math., to be published.

[22] P. Nayak and B. C. Williams, “Fast context switching in real-time propositional reasoning,” in *Proc. 14th Nat. Conf. Artif. Intell. (AAAI-97)*, 1997, pp. 50–56.

[23] D. S. Weld, “An introduction to least commitment planning,” *AI Mag.*, vol. 15, no. 4, pp. 27–61, 1994.

[24] G. Friedrich and W. Nejdl, “Choosing observations and actions in model-based diagnosis/repair systems,” in *Proc. 3rd Int. Conf. Principles Knowledge Representation Reasoning (KR-92)*, 1992, pp. 489–498.

[25] R. Fikes and N. Nilsson, “STRIPS: A new approach to the application of theorem proving to problem solving,” *Artif. Intell.*, vol. 2, pp. 189–208, 1971.

[26] R. J. Firby, “The RAP language manual,” Univ. Chicago, Chicago, IL, Animate Agent Project Working Note AAP-6, 1995.

[27] E. Gat, “ESL: A language for supporting robust plan execution in embedded autonomous agents,” in *Plan Execution: Problems Issues, Papers 1996 AAAI Fall Symp.*, 1996, pp. 59–64.

[28] J. Kurien, P. Nayak, and B. C. Williams, “Model-based autonomy for robust Mars operations,” presented at the 1st Int. Conf. Mars Soc., Boulder, CO, 1998.

[29] V. Gupta, R. Jagadeesan, and V. Saraswat, “Models of concurrent constraint programming,” in *Lecture Notes in Computer Science, CONCUR ’96: Concurrency Theory*. Heidelberg, Germany: Springer-Verlag, 1996, vol. 1119, pp. 66–83.

[30] M. Hofbaur and B. C. Williams, “Mode estimation of probabilistic hybrid systems,” in *Lecture Notes in Computer Science, Hybrid Systems: Computation and Control*. Heidelberg, Germany: Springer-Verlag, 2002, vol. 2289, pp. 253–266.

[31] B. C. Williams *et al.*, “Model-based reactive programming of cooperative vehicles for Mars exploration,” presented at the Int. Symp. Artif. Intell., Robotics, Automation Space (ISAIRAS-01), Montreal, QB, Canada, 2001.

Brian C. Williams (Member, IEEE) received the S.B., S.M., and Ph.D. degrees in computer science and artificial intelligence from the Massachusetts Institute of Technology (MIT), Cambridge, in 1981, 1984, and 1989, respectively.

From 1989 to 1994, he was with the Xerox Palo Alto Research Center, Palo Alto, CA, where he pioneered multiple-fault, model-based diagnosis in the 1980s and model-based autonomy in the 1990s through the Livingstone model-based health management and the Burton model-based execution systems. From 1994 to 1999, he was with the NASA Ames Research Center, Moffett Field, CA, where he formed the Autonomous Systems Area and coinvented the Remote Agent model-based autonomous control system. He was a member of the NASA Deep Space One flight team, which used Remote Agent to create the first fully autonomous, self-repairing explorer, demonstrated in flight in 1999. He was a member of the Tom Young Blue Ribbon Team in 2000, assessing future Mars missions in light of the Mars Climate Orbiter and Polar Lander incidents. He is currently a Professor at MIT, where he formed the Model-Based Embedded and Robotic Systems group, focusing on the development of model-based programming technology and its deployment to robotic explorers and cooperative systems.

Dr. Williams is currently a Member of the Advisory Council of the NASA Jet Propulsion Laboratory, Pasadena, CA. He received a NASA Space Act Award in 1999 for his work in the Autonomous Systems Area and with the Remote Agent control system.

Michel D. Ingham received the B.Eng. degree in mechanical engineering (honors program) from McGill University, Montreal, Canada, in 1995 and the S.M. degree from the Department of Aeronautics and Astronautics at the Massachusetts Institute of Technology (MIT), Cambridge, in 1998. He is a Ph.D. degree candidate at MIT.

In 1999, he spent a semester at the NASA Jet Propulsion Laboratory, Pasadena, CA, analyzing flight data from the Interferometry Program Experiment, a technology demonstration precursor for NASA’s Space Interferometry Mission. In 2001, he coordinated the MIT Model-Based Embedded and Robotic Systems group’s involvement in the New Millenium Program ST-7 autonomy concept definition study. He is currently a Research Assistant with the MIT Space Systems Laboratory, working with Prof. Brian Williams, and a Research Affiliate with the Artificial Intelligence Group, NASA Jet Propulsion Laboratory. His current research interests are in the field of model-based autonomy, focusing on hybrid model-based programming as an approach to robust sequencing and control of robotic spacecraft.

Mr. Ingham received the International Astronautical Federation’s 1998 Luigi G. Napolitano Award for significant contribution to aerospace research, for his research in the area of microdynamics of deployable space structures.

Seung H. Chung received the B.S. degree *magna cum laude* from the Department of Aeronautics and Astronautics Engineering at the University of Washington, Seattle, in 1999. He is completing a Master of Science degree at the Massachusetts Institute of Technology (MIT), Cambridge, and has been awarded a NASA Graduate Student Research Program fellowship for his doctoral studies.

He was in the Propulsion Flight Systems Group, NASA Jet Propulsion Laboratory, Pasadena, CA, participating in the design of the Mars Sample Return mission. In 2002, he worked with the Artificial Intelligence Group at the NASA Jet Propulsion Laboratory on the development of a distributed model-based executive. He is currently with the MIT Space Systems and Artificial Intelligence Laboratories, working with Prof. Brian Williams. His current research interests include model-based autonomy and its application to space missions, and the development of Titan, a model-based executive capable of autonomously estimating the state of spacecraft, diagnosing and repairing faults, and executing commands.

Paul H. Elliott received the S.B. degree in aeronautical and astronautical engineering and the S.B. degree in electrical engineering and computer science from the Massachusetts Institute of Technology (MIT), Cambridge, in 2001. He is currently working toward the S.M. degree in aeronautical and astronautical engineering at MIT, and intends to pursue the Ph.D. degree there.

He is currently a Research Assistant working with Prof. Brian Williams. His research interests include graphics, software engines, simulators, and compilation. He is currently working on model compilation.