

# Model-Based Programming of Fault-Aware Systems

*Brian C. Williams, Michel D. Ingham, Seung Chung,  
Paul Elliott, Michael Hofbaur, and Gregory T. Sullivan*

■ A wide range of sensor-rich, networked embedded systems are being created that must operate robustly for years in the face of novel failures by managing complex autonomic processes. These systems are being composed, for example, into vast networks of space, air, ground, and underwater vehicles. Our objective is to revolutionize the way in which we control these new artifacts by creating reactive model-based programming languages that enable everyday systems to reason intelligently and enable machines to explore other worlds. A model-based program is state and fault aware; it elevates the programming task to specifying intended state evolutions of a system. The program's executive automatically coordinates system interactions to achieve these states, entertaining known and potential failures, using models of its constituents and environment. At the executive's core is a method, called CONFLICT-DIRECTED A\*, which quickly prunes promising but infeasible solutions, using a form of one-shot learning. This approach has been demonstrated on a range of systems, including the National Aeronautics and Space Administration's Deep Space One probe. Model-based programming is being generalized to hybrid discrete-continuous systems and the coordination of networks of robotic vehicles.

## The Criticality of Fault-Aware Systems

The demands we place on robotic explorers and everyday embedded systems have gone through a major transformation over the last decade. For example, the challenge of robotic space exploration has dramatically shifted from simple planetary flybys to microrovers that can alight on several asteroids, collect the most

interesting geologic samples, and return with their findings. This challenge will not be answered through billion-dollar missions with 100-member ground teams but through innovation. Future space exploration will be enabled in significant part by inexpensive, "fire-and-forget" space explorers that are self-reliant and capable of handling unexpected situations; they must balance curiosity with caution.

Self-reliance of this sort can only be achieved through an explicit understanding of mission goals and the ability to reason from a model of how the explorer and its environment can support or circumvent these goals. This knowledge is used to carefully coordinate the complex network of sensors and actuators within the explorer. Given the complexity of current and future spacecraft, such fine-tuned coordination seems to be a nearly impossible task using traditional software engineering approaches.

Our demand for this level of fault resilience is no longer isolated to the realm of exotic space explorers. It has shifted to systems that are part of our everyday activities, such as our houses and automobiles (Williams and Nayak 1996b). Automobile manufacturers are now envisioning automobiles that address traffic congestion by operating cooperatively and that are perceived to never fail. Two decades of deregulation have placed much of the U.S.'s critical embedded infrastructure on open networks, including the telephone system, the internet, and power networks. Opening these systems exposes society to new levels of vulnerability to natural and manmade disasters.

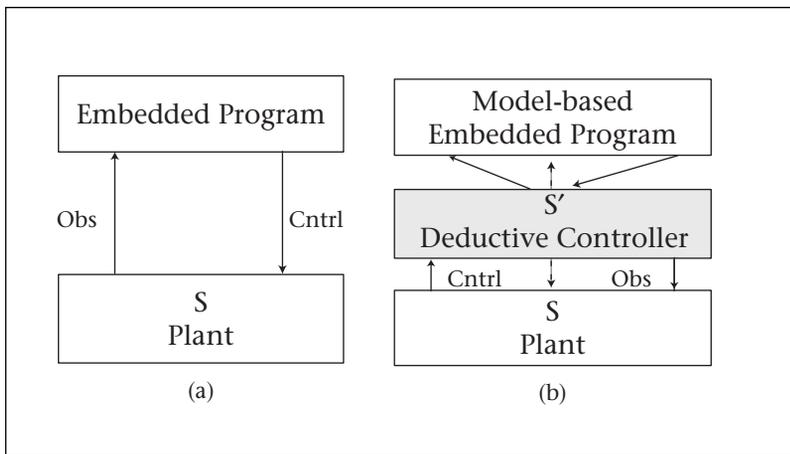


Figure 1. Models of Interaction.

A. Traditional embedded programs. B. Model-based programs.

Traditionally, closed systems are carefully protected only at their perimeter and are particularly vulnerable to failure or malicious attacks that originate within their perimeter. To be robust, open networked systems cannot leave their guard down along any front. They must quickly detect and recover from a malfunction in, or intrusion by, any of their constituents. This level of vigilance is common to space missions, which have repeatedly succeeded despite a multitude of hardware failures by using fault-management systems that continuously detect and recover from faulty components.

To achieve these new levels of vigilance, safety, and adaptivity, we must fundamentally rethink how we program embedded systems. We confront these challenges through an automated reasoning and programming paradigm called *model-based autonomy*. In this paradigm, embedded systems are easily programmed by specifying strategic guidance in the form of a few high-level control behaviors, called *model-based programs* (Williams et al. 2003) (see Programming in Terms of Hidden State). These control programs, along with a commonsense model of its hardware and its environment, enable an embedded system to control and monitor its hidden state according to the strategic guidance. To respond correctly in novel, time-critical situations, our systems use their onboard models to perform extensive commonsense reasoning within the reactive control loop, something that conventional AI wisdom had suggested was not feasible. Systems that execute model-based programs are called *model-based executives*. We focus on an executive, called TITAN, used to robustly coordinate the network of devices internal to a high-performance embedded system (see Model-Based Execution of Automatic Processes). At the core of

TITAN IS CONFLICT-DIRECTED A\*, a method for solving that quickly prunes away sets of candidate solutions, using a form of one-shot learning until the best feasible solution is found (see Efficient Online Reasoning through CONFLICT-DIRECTED A\*). In addition, we briefly highlight two other model-based executives: (1) MORIARTY, which is used to monitor, diagnose, and control hybrid discrete-continuous systems (see Model-Based Programming of Hybrid Systems), and (2) KIRK, which coordinates networks of robotic vehicles (see Model-Based Programming of Robotic Networks).

## Programming in Terms of Hidden State

Formal verification has long held promise for ensuring the robustness of embedded software. A major concern is the gap between the specifications about which we prove properties and the programs that are supposed to implement them. Manual translation across this gap introduces the danger of bugs. To close this gap, Berry (1989) emphasizes executable specifications within the ESTEREL embedded language: “What you prove is what you execute.” In model-based programming, we carry the idea of executable specification one step further by offering an executable specification language that operates on descriptions of abstract hidden states, reasons through physical models in real time, and is knowledgeable of a system’s fault behavior.

Engineers like to reason about embedded systems in terms of state evolutions, providing the engineer with a simple abstraction that ignores issues of controllability and observability. However, executable specification languages, such as ESTEREL and STATECHARTS (Harel 1987), interact with a physical plant by reading sensor variables and writing control variables (figure 1a). It is the programmer’s responsibility to close the gap between intended state and the sensors and actuators. This mapping involves reasoning through a complex set of interactions under a range of possible failure situations. The complexity of the interactions and the number of possible scenarios make this process error prone.

A *model-based programming language* is an executable specification language similar to ESTEREL or STATECHARTS but with the additional feature that it is *state aware*; that is, it interacts directly with the plant state (figure 1b). This is accomplished by allowing the programmer to read or write constraints on “hidden” state variables in the plant, that is, states that are not directly observable or controllable. It is then

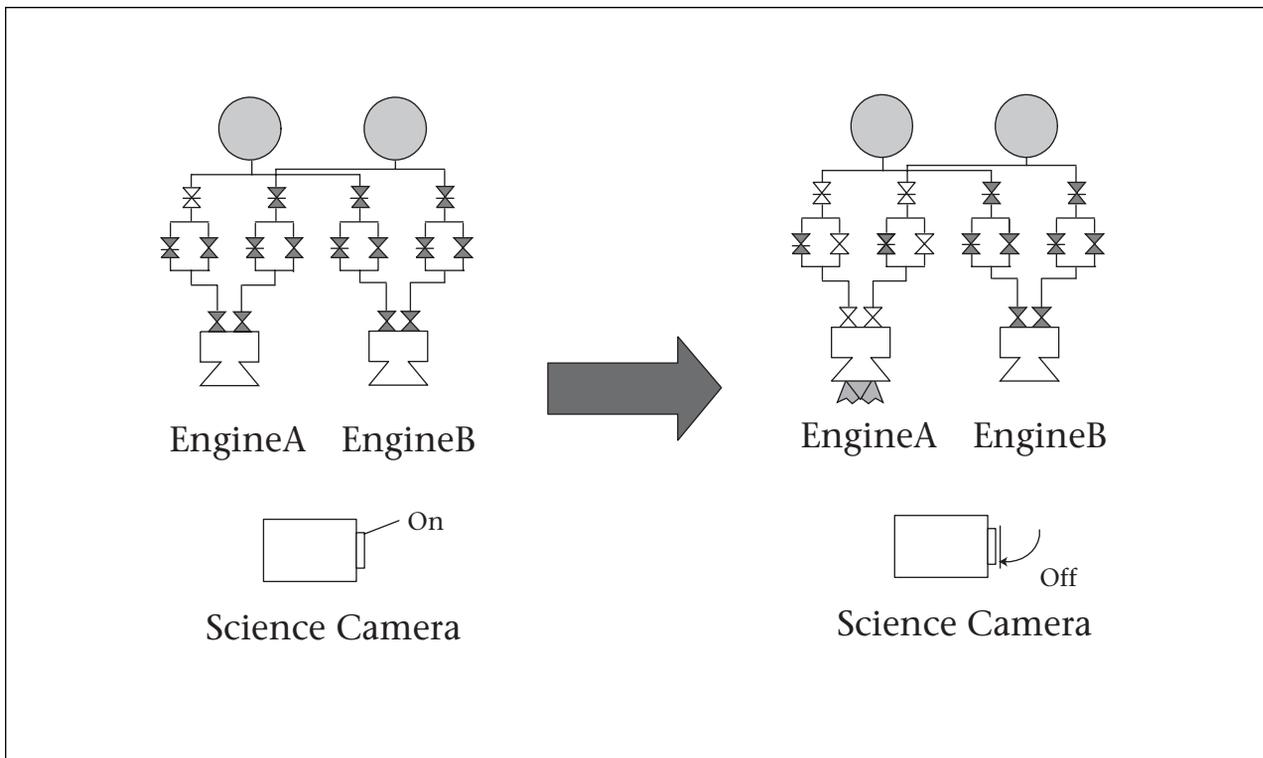


Figure 2. Simple Spacecraft for Orbital Insertion.

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 using a deductive controller that reasons from a commonsense plant model. To be robust, this mapping must succeed under failure; hence, the deductive controller must reason extensively from models of correct and faulty component failure. Given the exponential space of potential symptoms, diagnoses, and recoveries, some of this reasoning must be performed on-line. In this article, we introduce the REACTIVE MODEL-BASED PROGRAMMING LANGUAGE (RMPL), a programming framework that supports model-based execution from hybrid systems to robotic networks.

## Model-Based Programming

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

ism, called *probabilistic concurrent constraint automata*, unifies constraints, concurrency, and Markov processes.

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 (figure 2). An engineer thinks about this maneuver in terms of state trajectories:

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

This specification is far simpler than a control program that must turn on heaters and valve drivers, open valves, and interpret sensor readings for the engine. Thinking in terms of more abstract hidden states makes the task of writing the control program much easier and avoids the error-prone process of reasoning through low-level system interactions. In addition, it gives the program's execution kernel the latitude to respond to failures as they arise, which is essential for achieving high levels of robustness.

Next, consider a model-based program corresponding to this specification. The spacecraft dual main engine system (figure 2) consists of

```

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

```

Figure 3. RMPL Control Program for Orbital Insertion.

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. An accelerometer sensor, for example, is used to confirm engine operation by sensing thrust, and a camera shutter position sensor is used to confirm camera operation.

### Control Program

The RMPL control program (figure 3) codifies the informal specification we gave earlier as a set of state trajectories. RMPL provides standard embedded programming constructs, such as parallel and sequential execution, iteration, conditions, and preemption. Recall that to perform orbital insertion, one of the two engines must be fired. We start by concurrently placing the two engines in standby and shutting off the camera, which is performed by lines three to five; the comma at the end of each line denotes parallel composition. We then fire an engine, choosing to use engine *A* as the primary engine (lines 6–9) and engine *B* as a backup in the event that engine *A* fails to fire correctly (lines 10–11). Engine *A* starts trying to fire as soon as it achieves standby, and the camera is off (line 7) but aborts if at any time engine *A* is found to be in a failure state (line 9). Engine *B* starts trying to fire only if engine *A* has failed, *B* is in standby, and the camera is off (line 10).

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 as both assertions and execution conditions. For example, in lines six to nine, EngineA = Firing appears in an assertion (line 8), and 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; that is, only shutter position and acceleration can directly be sensed, and only the flight computer command can directly be set. Finally, by referring to hidden states directly, the RMPL program is far simpler than the corresponding traditional program, which operates on sensed and controlled variables. The added complexity of the traditional program is because of the need to fuse sensor information and generate command sequences under a large space of possible operation and fault scenarios.

For example, consider how a traditional program achieves the lone assignment EngineA = Firing. From a large space of options, the program must first select a set of healthy valves whose opening will achieve thrust. To select the appropriate valves, the program encodes a decision tree that assesses the health of the valves by fusing multiple sources of sensor data. Next, the selected valves are opened using a valve-open procedure. This procedure must send commands over a communication bus to a valve driver, which then opens the valve. The procedure involves another decision tree that is able to detect and recover from any failures along this path, again by exploiting redundancy.

This example demonstrates that the code needed to achieve even a simple hidden state assignment can quickly explode. Writing this

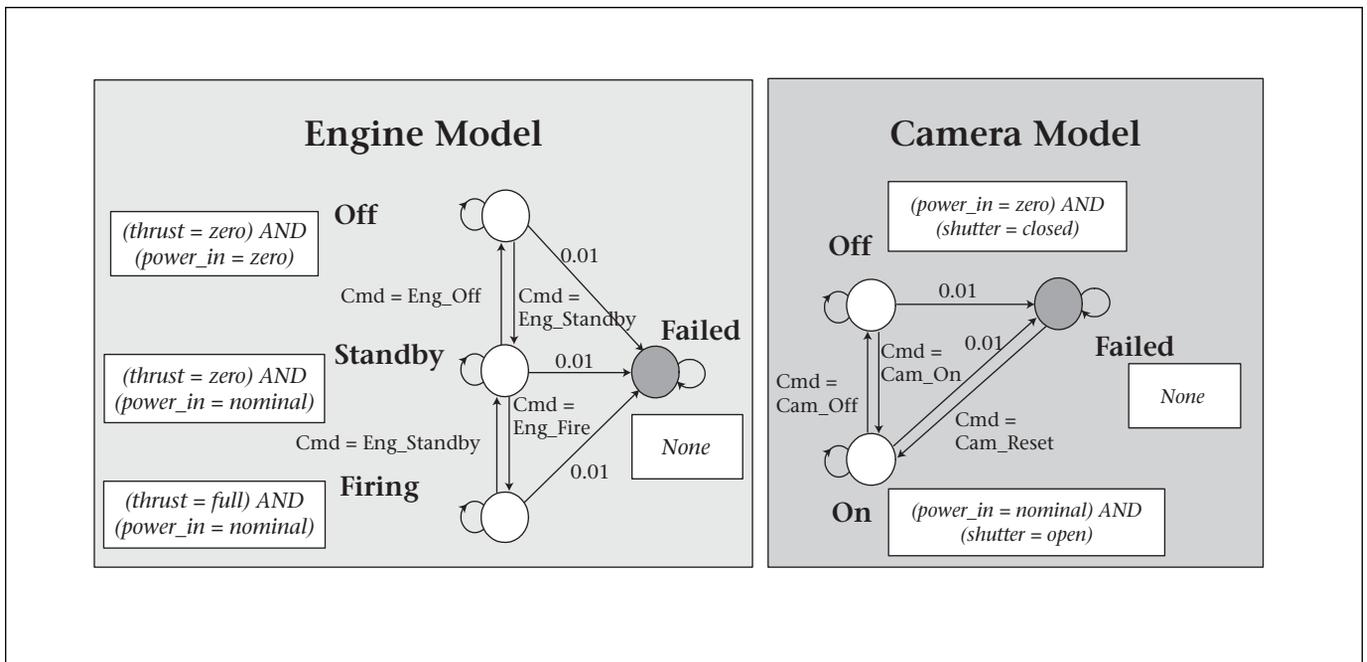


Figure 4. State-Transition Models for a Simplified Spacecraft.

type of code is tedious. In this situation, the programmer can inadvertently introduce a software bug or miss an important case that puts the mission at risk. For example, the Mars Polar Lander mission was most likely lost when a buggy software monitor incorrectly classified a noise spike on one of the lander's legs as an indication of touchdown. The lander then shut off its engine roughly 40 meters above the surface (Young et al. 2000). In contrast, in model-based programming, the control program is a compact specification of intended state evolution that is executed by provably correct synthesis procedures, using knowledge of failure provided by a compact, reusable plant model.

### Plant Model

The plant model represents a system's normal behavior and its known and unknown aberrant behaviors. It is used by a deductive controller to map sensed variables to queried states in the control program and asserted states to specific control sequences. The plant model is specified as a concurrent transition system, composed of probabilistic concurrent constraint automata (Williams and Nayak 1996a). Each component automaton is represented by a set of component modes, a set of constraints defining the behavior within each mode, and a set of probabilistic transitions between modes. Constraints are used to represent cotemporal interactions between state variables and inter-communication between components. Constraints on continuous variables operate on

qualitative abstractions of the variables, comprised of the variable's sign (positive, negative, zero) and deviation from nominal value (high, nominal, low). Probabilistic transitions are used to model the stochastic behavior of components, such as failure and intermittency. Reward is used to assess the costs and benefits associated with particular component modes. The component automata operate concurrently and synchronously.

For example, we can model the spacecraft abstractly as a three-component system (two engines and a camera) by supplying the models depicted graphically in figure 4. Nominally, an engine can be in one of three modes: (1) off, (2) standby, or (3) firing. The behavior within each of these modes is described by a set of constraints on plant variables, namely, thrust and power\_in. In figure 4, these constraints are specified in boxes next to their respective modes. The engine also has a failed mode, capturing any off-nominal behavior. We entertain the possibility of a novel engine failure by specifying no constraints for the engine's behavior in the failed mode (Davis 1984).

A wide range of plant-modeling formalisms is possible, depending on the category of system being controlled. These formalisms define different families of model-based programming languages. For example, in Hofbaur and Williams (2002), the plant models are represented as a hybrid between hidden Markov models (HMMs) and continuous dynamics to detect and handle incipient failures. In

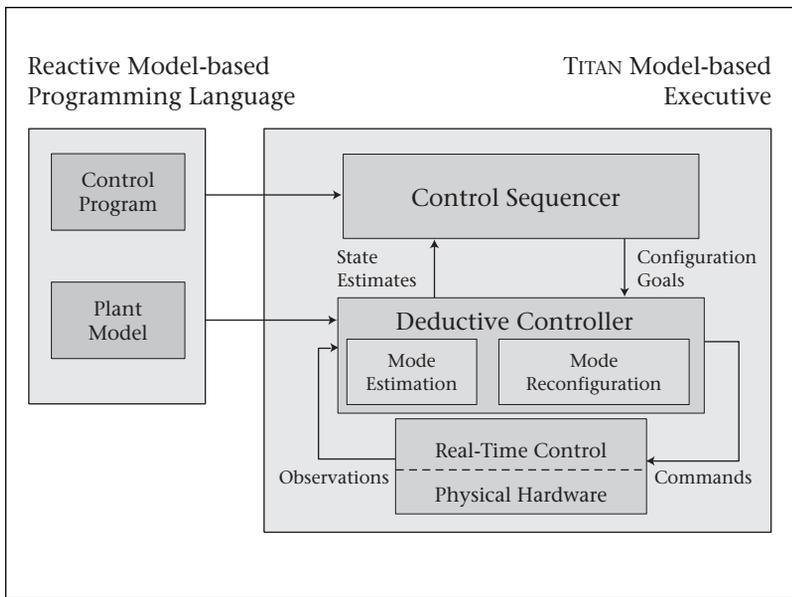


Figure 5. Architecture for a Model-Based Executive.

Williams, Chung, and Gupta (2001), the plant models are described using a hybrid of hierarchical automata and constraints to monitor robotic networks.

## Model-Based Execution of Autonomous Processes

A model-based program is executed by automatically generating a control sequence that moves the physical plant to the states specified by the control program (figure 5). We call these specified states *configuration goals*. Program execution is performed using a *model-based executive*, which generates configuration goals and then generates a sequence of control actions that achieve each goal based on knowledge of the current plant state and model.

A model-based executive consists of two components: (1) a control sequencer and (2) 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 issuing commands to move the plant through a sequence of states that achieve the goals (mode reconfiguration).

Consider a model-based executive, called TITAN, which coordinates the low-level autonomous processes internal to an embedded system. When TITAN executes the orbital insertion control program (figure 3), the control sequencer

starts by generating a configuration goal consisting of the conjunction of three state assignments: (1) EngineA = Standby, (2) EngineB = Standby, and (3) Camera = Off (lines 3–5). To determine how to achieve this goal, the deductive controller considers the latest estimate of the state of the plant. For example, suppose the deductive controller determines from its sensor measurements and previous commands that the two engines are already in standby, but the camera is on. The deductive controller deduces from the model that it should send a command to the plant to turn the camera off. After executing this command, it uses its shutter position sensor to confirm that the camera is off. With Camera = Off and EngineA = Standby, the control sequencer advances to the configuration goal of EngineA = Firing (line 8). The deductive controller identifies an appropriate setting of valve states that achieves this behavior, then it sends out the appropriate commands.

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

### Mode Estimation

*Mode estimation* incrementally tracks the set of state trajectories that are consistent with the plant model, the sequence of observations, and control actions. For example, suppose the deductive controller is trying to maintain the configuration goal EngineA = Firing, as shown to the left in figure 6. Here, we assume that mode estimation starts with knowledge of the initial state, with valves opening a flow of oxidizer and fuel to engine A. In the next time instant, the sensors send back the observation that Thrust = zero. Mode estimation then identifies a number of state transitions that are consistent with this observation, including either the inlet valve into engine A has transitioned to stuck closed, as depicted on the right in figure 6, or any combination of valves along the fuel or oxidizer supply path is broken.

We frame mode estimation as an instance of belief state update for an HMM. It incremental-

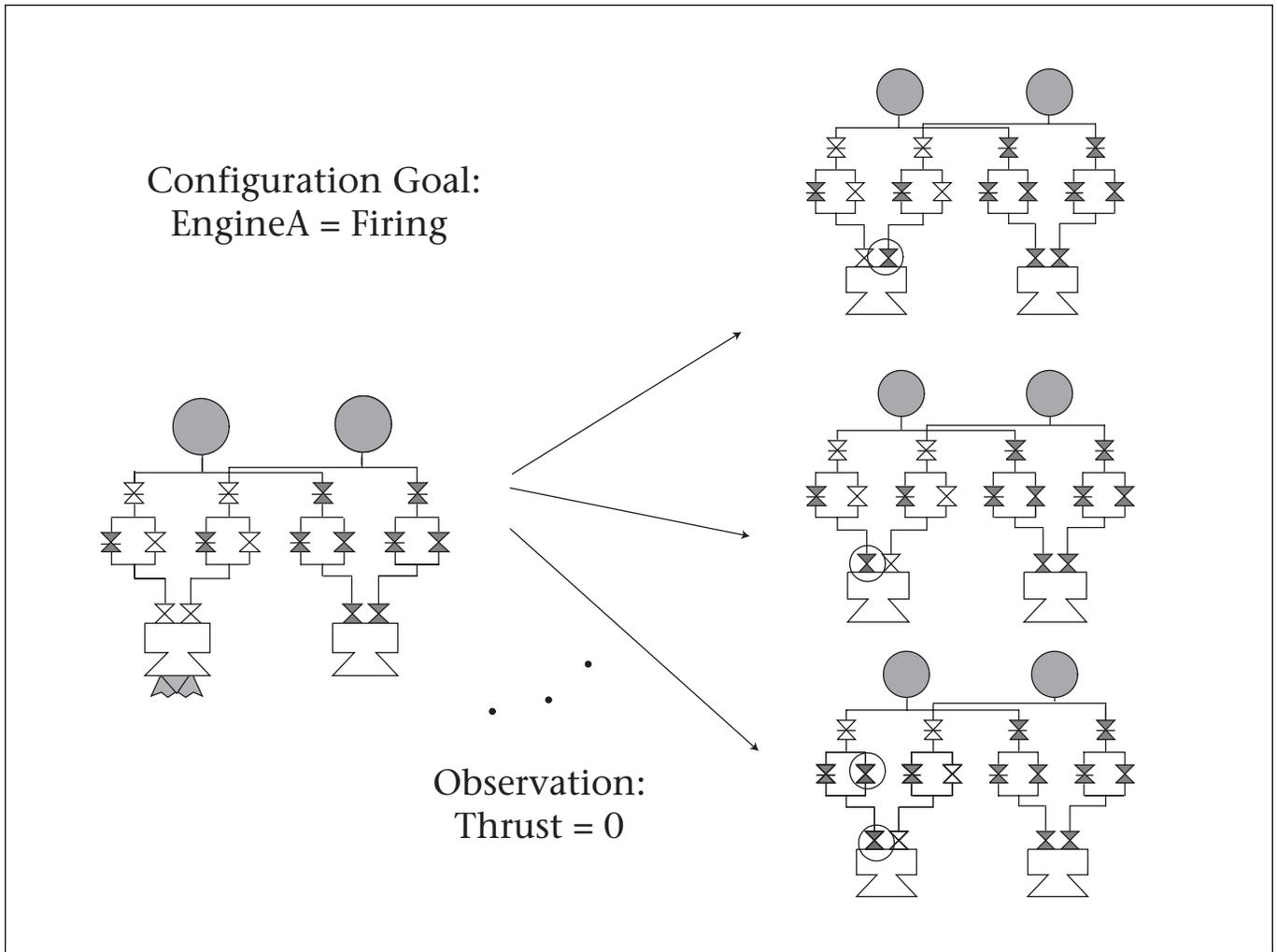


Figure 6. Mode Estimation Step for Orbital Insertion.

Faulty valves are circled, and closed valves are filled.

ly computes the probability of state  $s_i$  at time  $t+1$  using a combination of forward prediction from the model and correction based on the observations

$$p^{(\bullet t+1)}[s_i] = \sum_{j=1}^n p^{(\bullet t)}[s_j] P_{\top}(s_i | s_j, \mu^{(t)})$$

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

where  $P_{\top}(s_i | s_j, \mu^{(t)})$  is defined as the probability that the plant transitions from state  $s_j$  to state  $s_i$ , given control actions  $\mu^{(t)}$  and  $P_{\circ}(o_k | s_i)$  is the probability of observing  $o_k$  in state  $s_i$ . Probability  $p^{(\bullet t+1)}[s_i]$  is conditioned on all observations to  $o^{(t)}$ , but  $p^{(t+1 \bullet)}[s_i]$  is also conditioned on the latest observation  $o^{(t+1)} = o_k$ .

Our approach is distinguished in that the plant HMM is encoded compactly through concurrency and constraints. The number of

states is exponential in the number of components, which reaches a trillion states for even our simple example. Hence, mode estimation enumerates the consistent trajectories and states in order of likelihood using an efficient procedure called CONFLICT-DIRECTED A\*, described later. Mode estimation offers an anytime approach, which stops enumeration when no additional computational resources are available.

### Mode Reconfiguration

*Mode reconfiguration* takes as input a configuration goal  $g^{(t)}$ , and the most likely current state  $s^{(t)}$  computed by mode estimation, and returns a series of commands that progress the plant toward a maximum-reward goal state that achieves  $g^{(t)}$  (Williams and Nayak 1997). Mode reconfiguration accomplishes this using a goal interpreter and a reactive planner. A *goal inter-*

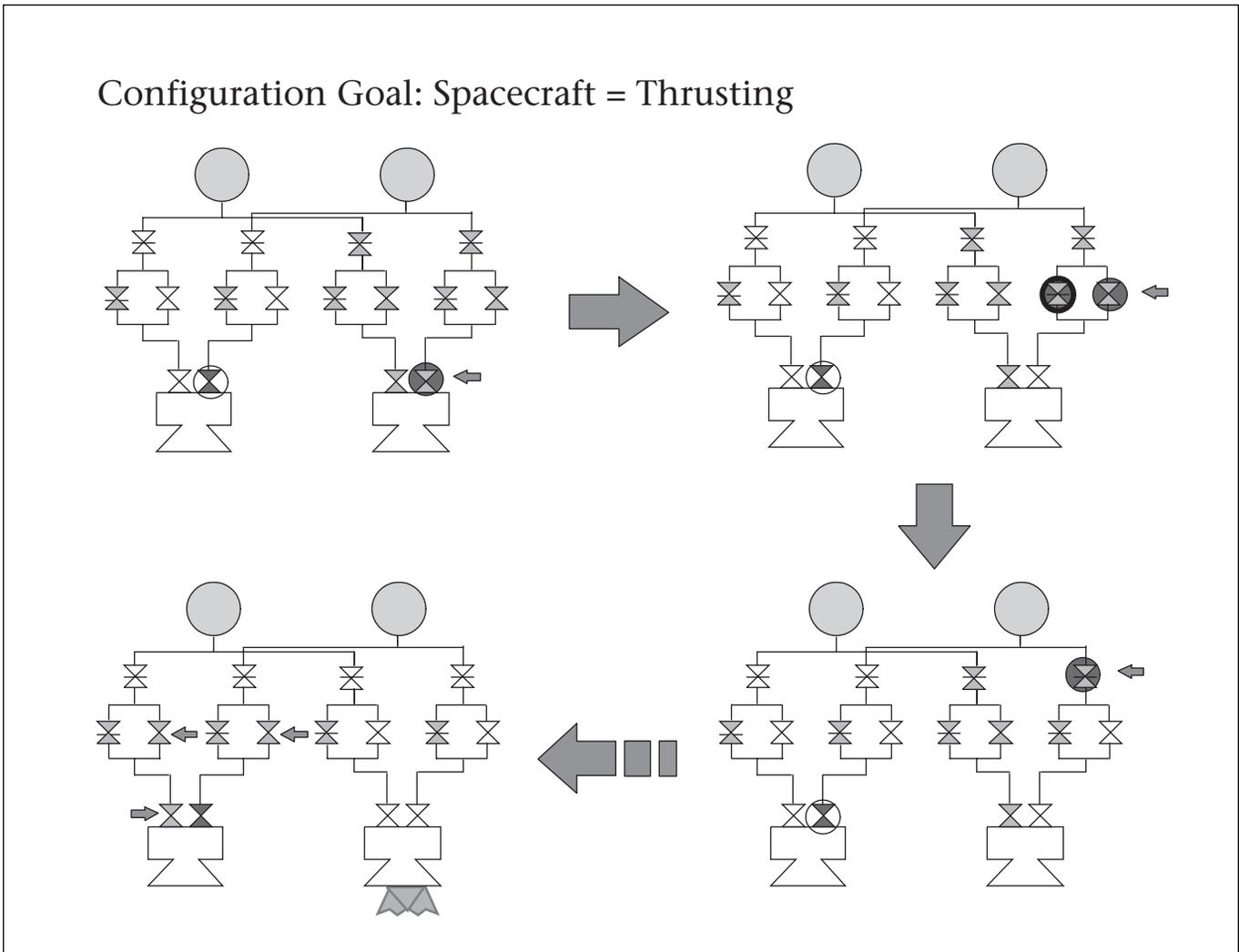


Figure 7. The Goal Interpreter Uses CONFLICT-DIRECTED A\* to Search for a Mode Reconfiguration during Orbital Insertion..

preter determines a target state  $s_g^{(t)}$  that is reachable from  $s^{(t)}$  and achieves  $g^{(t)}$ , maximizing reward. It accomplishes this by having CONFLICT-DIRECTED A\* search over the reachable states in best-first order. A reactive planner generates a command sequence that moves the plant from  $s^{(t)}$  to  $s_g^{(t)}$ . A 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 EngineB = Firing, the goal interpreter selects a set of valves to open that establish a flow of fuel to the engine (bottom left, figure 7). The reactive planner sends commands to control units, drivers, and valves to achieve this target.

### Model-Based Reactive Planning

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. In fact, this is how TITAN's predecessor, LIVINGSTONE (Williams and Nayak 1996a), 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. For example, figure 8 shows the communication paths to a spacecraft main engine system. The flight computer sends commands to a bus controller, which broadcasts these commands over a 1553 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.

A robust close-valve routine should be able to handle the following example scenario. The corresponding procedure is automatical-

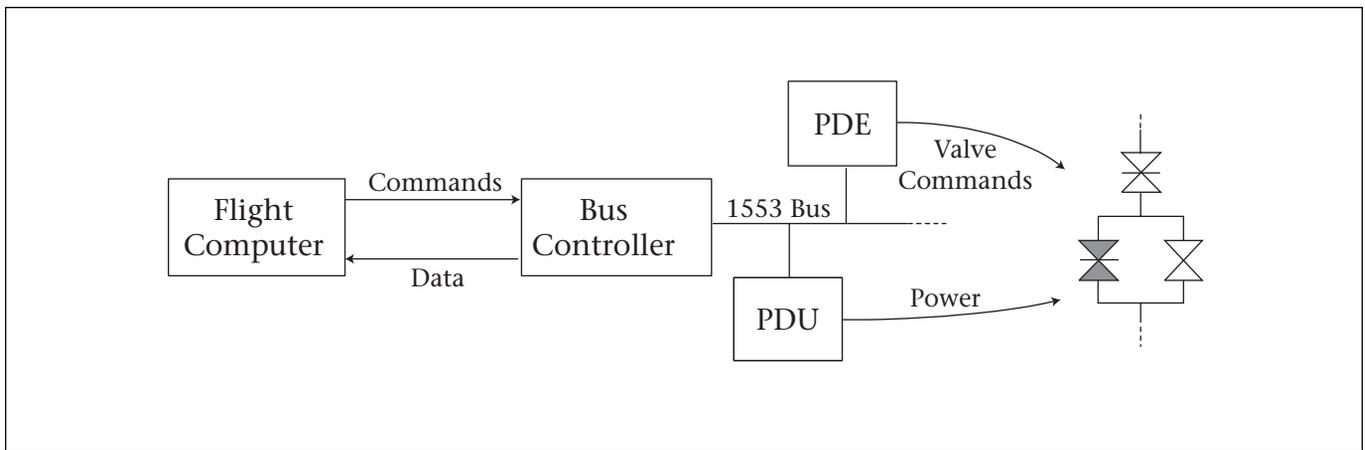


Figure 8. A Spacecraft Establishes Complex Interaction Paths from the Flight Computer to the Main Engine.

ly generated by TITAN's reactive planner:

To ensure that a valve is closed, the close-valve routine first ascertains if the valve is open or closed by polling its sensors. If it is open, it broadcasts a close command. However, first it determines if the driver for the valve is on, again by polling its sensors, and if not, it sends an on command. Now suppose that shortly after the driver is turned on, it transitions to a resettable failure mode. The valve routine catches this failure, and then before sending the close command, it issues a reset command. Once the valve has closed, the close-valve routine powers off the valve driver to save power. However, before doing so, it changes the state of any other valve that is controlled by that driver; otherwise, the driver needs to be immediately turned on again, wasting time and power.

This example highlights several key challenges: Devices are controlled indirectly through other devices; they can negatively interact and, hence, need to be coordinated; they fail and, hence, need to be monitored; and when they fail, they must quickly be repaired. To address these challenges, TITAN's reactive planner precompiles a set of procedures that form a goal-directed universal plan, specifying responses for achieving all possible target states, starting in all possible current states. These procedures constitute a set of compact concurrent policies, one for each component, and are generated by exploiting properties of causality and reversibility of action. In contrast, the size of an explicit universal plan is exponential in the number of components. TITAN's reactive planner, called BURTON, is developed in Williams et al. (2003) and Williams and Nayak (1997).

## Efficient Online Reasoning through CONFLICT-DIRECTED A\*

The core problems underlying model-based programming, such as mode estimation and goal interpreter, involve a search over a discrete space for the best solution that satisfies a set of finite-domain constraints. These problems, called *optimal constraint-satisfaction problems* (OCSPs), consist of a set of decision variables  $y$ , each ranging over a finite domain, a utility function  $f$  on  $y$ , and a set of constraints  $C$  that  $y$  must satisfy. A solution is an assignment to  $y$  that maximizes  $f$  and satisfies  $C$ . For mode estimation, each  $y$  denotes a set of possible next transitions for a component;  $f$  maximizes transition probability; and  $C$  denotes consistency between the target state, model, and observations. For a goal interpreter, each  $y$  denotes sets of reachable transitions for a component,  $f$  minimizes target state cost, and  $C$  denotes the entailment of the configuration goal by the target state. A key to the success of model-based programming is the ability to perform this search quickly and correctly. The best methods for finding optimal solutions, such as A\*, explore the space of solutions one state at a time, visiting every state whose estimated utility is greater than the true optimum (figure 9a). The time taken to visit this number of states is unacceptable for model-based executives, which perform best-first search within the reactive control loop.

The CONFLICT-DIRECTED A\* method, used by TITAN, also searches in best-first order but accelerates search by eliminating subspaces around each state that are inconsistent. This process builds on the concepts of conflict and kernel diagnosis introduced in model-based diagnosis (de Kleer and Williams 1987; de Kleer, Mack-

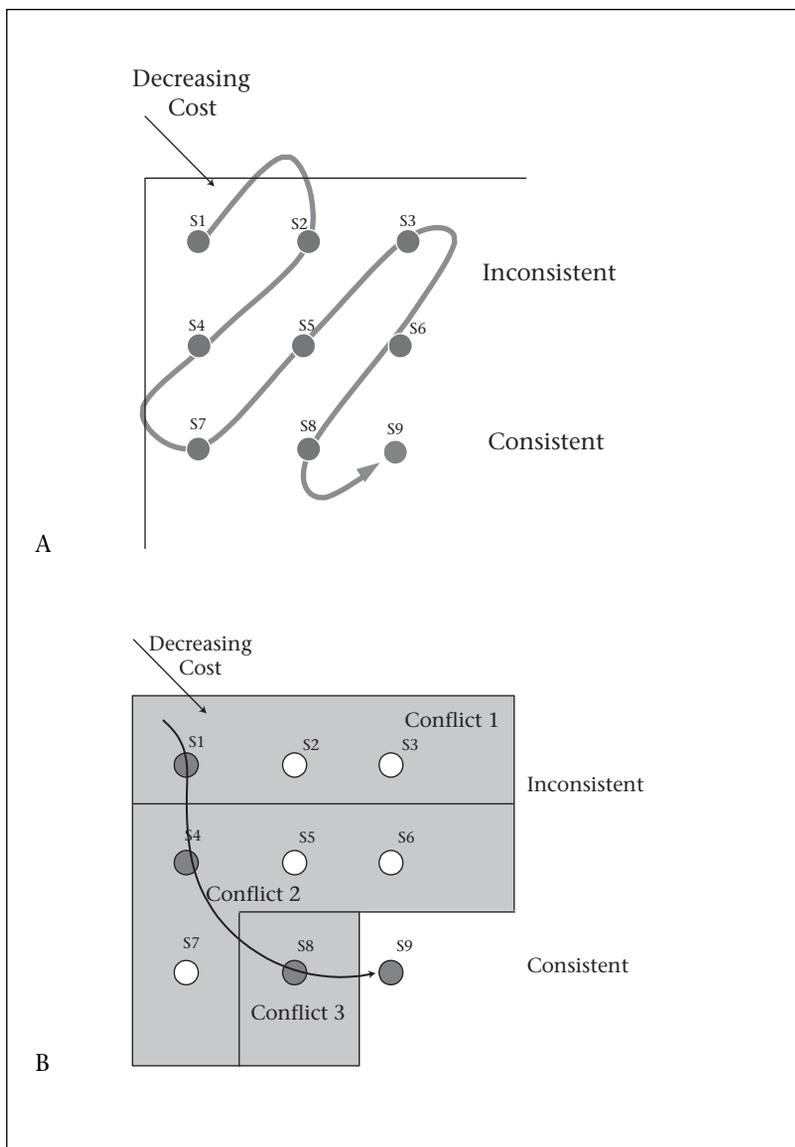


Figure 9. A\* and CONFLICT-DIRECTED A\*.

A. A\* examines all best-cost states to the solution. B. CONFLICT-DIRECTED A\* jumps over best-cost states contained by known conflicts.

worth, and Reiter 1992). A *conflict* describes a set of states that are inconsistent with the constraints. A state contained by a conflict manifests the conflict, and a state not contained by a conflict resolves the conflict. The kernel diagnoses describe a set of states that resolve all known conflicts and, hence, the portion of the search space that hasn't yet been pruned.

In figure 9b, CONFLICT-DIRECTED A\* first selects state S1, which proves inconsistent. This inconsistency generalizes to Conflict 1, which eliminates states S1 to S3. CONFLICT-DIRECTED A\* then tests S4 as the highest utility state resolving Conflict 1. S4 tests inconsistent and generates Conflict 2, eliminating states S4 to S7. CONFLICT-DIRECTED A\* continues until it finds S9

consistent and, hence, an optimal solution. Figure 7 shows a similar conflict-detection sequence, generated by a goal interpreter for the Cassini example.

CONFLICT-DIRECTED A\* consists of four steps: First, a candidate  $S$  is generated, which is the best-valued decision state that resolves all discovered conflicts. Second,  $S$  is tested for consistency against the constraints. Third, when  $S$  tests inconsistent, the inconsistency is generalized to one or more conflicts, denoting states that are inconsistent in a manner similar to  $S$ . Fourth, CONFLICT-DIRECTED A\* jumps down to the next-best candidate  $S'$  that resolves all conflicts discovered thus far. This process repeats until the desired leading solutions are found, or all states are eliminated. (Note that the candidate is tested using any suitable CSP algorithm that extracts conflicts, allowing CONFLICT-DIRECTED A\* to be applied to a wide family of constraint systems).

The CONFLICT-DIRECTED A\* algorithm is presented in Williams and Ragno (2003) and makes rigorous use of a set of focusing mechanisms first introduced heuristically within the SHERLOCK diagnosis system (deKleer and Williams 1989) and evolved within the model-based-diagnosis community over the last decade. Although it emerged in the context of diagnosis, we have found CONFLICT-DIRECTED A\* to be an equally powerful algorithm for reconfiguration, planning, and knowledge compilation.

## Model-Based Execution Six Light Minutes from Earth

TITAN and its predecessors, LIVINGSTONE (Williams and Nayak 1996a), SHERLOCK (deKleer and Williams 1989), and GDE (de Kleer, Mackworth, and Reiter 1987), have been applied to a wide range of applications over the last two decades, including space systems, copiers, automobiles, electronics, power transmission systems, and biological systems. In the space domain, we are currently incorporating RMPL and TITAN within the Massachusetts Institute of Technology (MIT) SPHERES multi-spacecraft mission, which is on the manifest to be flown inside the International Space Station. In addition, we are working in collaboration with the California Institute of Technology Jet Propulsion Laboratory to apply RMPL and TITAN to the National Aeronautics and Space Administration (NASA) Mars 2009 Rover mission. TITAN has also been applied to test beds for the United States Air Force TechSat 21 mission and NASA's Messenger and Space Technology 7 missions.

TITAN's deductive controller is a generalization of the LIVINGSTONE mode estimation and re-configuration system (Williams and Nayak 1996a). LIVINGSTONE was demonstrated in flight in the spring of 1999 on NASA's New Millennium Deep Space One (DS1) probe as part of the remote-agent autonomy experiment (Bernard et al. 1999). DS1 is an asteroid and comet flyby mission that used an ion-propulsion system and navigated by the stars using a camera and on-board star map. The remote-agent experiment demonstrated that an autonomous system can automatically plan and execute a space mission, given a set of mission goals and a spacecraft operation model, and that it can recover from failures by diagnosing and repairing the spacecraft using engineering models. During this experiment, LIVINGSTONE was exercised on a wide range of failures: It detected that a camera was stuck on and invoked mission replanning to handle the loss of resources, it detected a switch sensor failure and determined that it was harmless, it repaired an instrument by issuing a reset, and it compensated for a stuck-closed thruster valve by switching to a secondary control mode.

The thruster scenario involved isolating a faulty valve among eight valve-thruster pairs but only sensing a three-dimensional acceleration. A model using only the sign of quantities and their relative value was sufficient for LIVINGSTONE to perform this task. In addition, because of the simplicity of these models, their development time was a minor fraction of the total development time for the remote-agent experiment.

Beyond the space domain, TITAN is being demonstrated in the context of two automotive applications. The first application is the control and fault management of a multivehicle cooperative cruise control system developed at University of California at Berkeley. The second application is an on-board automobile fault-management system, which is being developed in collaboration with Toyota.

RMPL offers one instance of a larger family of reactive model-based executive programming languages, which are parameterized by the plant modeling language and its corresponding deductive controller. The next two sections highlight two different variants of RMPL.

## Model-Based Programming of Hybrid Systems

The year 2000 was kicked off with two missions to Mars, following on the heels of the highly successful Mars Pathfinder mission. The Mars Climate Orbiter burned up in the Martian

atmosphere when a units error in a small force table introduced a small but indiscernible fault that, over a lengthy time period, caused the loss of the orbiter. The problem of misinterpreting a system's dynamics was punctuated later in the year when the Mars Polar Lander vanished. It most likely crashed into Mars after it incorrectly shut down its engine 40 meters above the surface because it misinterpreted its altitude as a result of a faulty software monitor.

This case study is a dramatic instance of a common problem—increasingly complex systems are being developed whose failure symptoms are nearly indiscernible until a catastrophic result occurs. In addition, these failures are manifested through a coupling between a system's continuous dynamics and its evolution through different behavior modes.

We address these issues through a hybrid model-based executive called MORIARTY whose mode-estimation capability is able to track a system's behavior along both its continuous state evolution and its discrete mode changes (Hofbauer and Williams 2002). Failures can generate symptoms that are initially on the same scale as sensor and actuator noise. To discover these symptoms, MORIARTY uses statistical methods to separate the noise from the true dynamics.

MORIARTY extends TITAN's concurrent probabilistic constraint automata model to include continuous dynamic system models as constraints (top left, figure 10). This framework is unlike most traditional hybrid modeling systems (for example, Henzinger [1996]) that define mode transitions to be deterministic or do not explicitly specify probabilities for transitions. MORIARTY tracks a system's hidden state by creating a hybrid HMM observer (left, figure 10). The observer uses the results of continuous-state estimates to judge a system's mode changes and coordinates the actions of a set of continuous-state observers. This approach is similar to work pursued in multimodel estimation (Li and Bar-Shalom 1996; Maybeck and Stevens 1991). However, MORIARTY provides a novel anytime, any-space algorithm for computing approximate hybrid estimates, which allows it to track concurrent automata that have a large number of possible modes. MORIARTY is being demonstrated on Mars entry, descent, and landing scenarios (right, figure 10) and on the fault management of a simulated Martian habitat.

## Model-Based Programming of Robotic Networks

Thus far, we have focused on model-based programming methods that increase robustness and autonomy by generating the autonomic

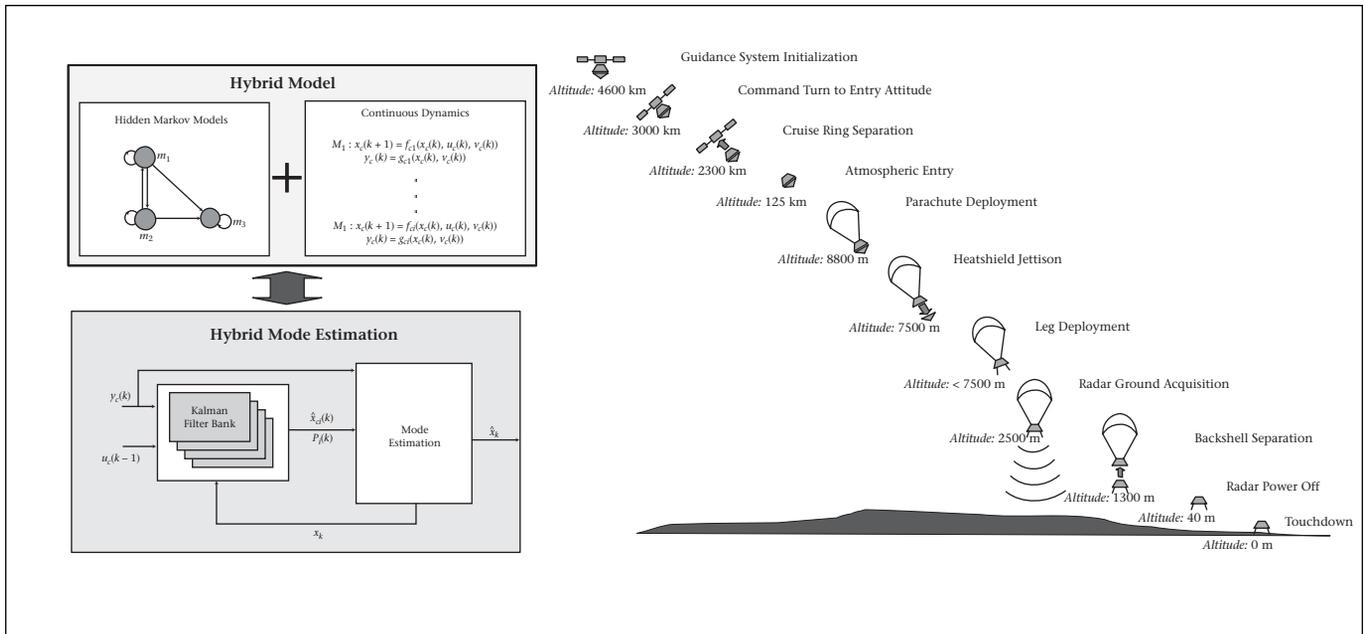


Figure 10. Hybrid Estimation Architecture and Its Potential Application. Left: Hybrid mode estimation. Right: Mars entry, descent, and landing sequence (courtesy, NASA JPL).

processes internal to embedded systems, such as spacecraft and automobiles. The future looks to the creation of cooperative robotic networks in which robotic systems act together to achieve elaborate missions within uncertain environments. This network can be a heterogeneous collection of planes, helicopters, boats, and ground vehicles that perform search and rescue during natural or manmade disasters or a set of rovers, blimps, and orbiters that explore science sites on Mars (figure 11). For example, to explore Mars, an orbiter performs initial surveillance, producing a coarse site map. An agile scout rover, with a tethered blimp, refines the map with high-resolution data for local regions and performs initial evaluation of the scientific sites. Finally, a laboratory rover performs detailed evaluation of scientifically promising sites.

To program these robotic networks quickly and robustly, we extend model-based programming with constructs for specifying global strategies for multivehicle coordination. We refer to this extended language as cooperative RMPL and its corresponding executive as KIRK (Kim, Williams, and Abramson 2001). KIRK generalizes TITAN’s capabilities for deducing and controlling hidden state by including capabilities for reasoning about contingencies, scheduling activities, and planning cooperative paths. To support this reasoning, RMPL’s plant model is extended to include models of the external environment, such as a terrain map, and specifications for the command set and dy-

namics of each robot vehicle in the network. Most embedded programming languages and robotic execution languages, such as outlined in Firby (1995), use myopic execution strategies that do not evaluate their future course of action in terms of feasibility or optimality. KIRK is distinguished in that it first “looks” by using fast temporal planning methods that identify the optimal consistent strategy within the RMPL program. The result is a partially ordered temporal plan. KIRK then “leaps” using a robust plan-execution algorithm, described in Tsamardinos, Muscettola, and Morris (1998), which adapts to execution uncertainties through fast, online scheduling.

To look, KIRK’s control sequencer chooses a set of execution threads from a nondeterministic RMPL program, producing a configuration plan, and checks to ensure that this plan is consistent and can be scheduled. The plan comprises temporally bounded configuration goals that specify desired states. KIRK’s deductive controller uses the plant model to map the configuration plan to a plan that involves executable robot commands. Both the control sequencer and the deductive controller utilize variants of graph-based temporal planning to accelerate reasoning.

The Mars exploration concept has been validated within the MIT cooperative robotics test bed using four ATRV rovers and an overhead stereo camera, emulating a blimp (figure 12) (Williams et al. 2001). KIRK has also been demonstrated in simulation on the coordina-



Figure 11. Mars Exploration Using Rovers.

tion of as many as nine air vehicles performing a suppression of enemy air defense mission.

## Discussion

Sensor-rich, networked embedded systems are taking the world by storm, from our everyday automobiles to futuristic robotic networks. In this article, we argued that a radically different programming paradigm is needed, one that frees the programmer from the myriad details of managing low-level interactions and detailed failure analysis. Our solution—model-based programming—allows the programmer to elevate his/her thinking to the level of specifying intended state evolutions, relinquishing issues of sensing and control to the language's model-based executive.

Through the TITAN executive, we demonstrated how model-based programming can be used to easily generate the autonomic processes internal to robust embedded systems. The early sibling of TITAN's deductive controller, LIVINGSTONE, was demonstrated to diagnose and repair a half dozen failures in flight on the NASA New Millennium DS1 probe. TITAN is currently

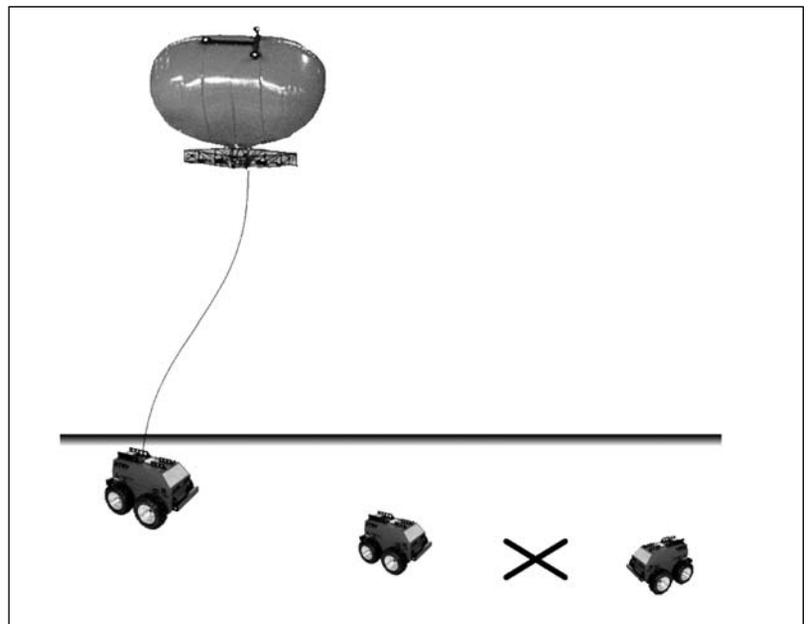


Figure 12. Mars Exploration Using Blimps.

being demonstrated on automobiles, Mars rovers, and a microspacecraft cluster within the International Space Station. We are extending TITAN with methods for compile-time synthesis and verification of model-based programs.

Through the MORIARTY executive, we are demonstrating how the model-based-programming paradigm can control high-fidelity systems using hybrid models, enabling the detection of subtle failures and the creation of high-confidence systems, such as Mars entry descent and landing codes. Through the KIRK executive, we are demonstrating how the model-based-programming paradigm is expanding to networked embedded systems whose components are highly capable, agile robots.

### Acknowledgments

We would like to thank Vineet Gupta, whose early collaboration and work on timed and hybrid concurrent constraint languages has deeply influenced this work. We would particularly like to thank the rest of the Model-based Embedded and Robotic Systems team for their extensive insights and efforts in the creation of RMPL, TITAN, MORIARTY, and KIRK: Mark Abramson, Judy Chen, Lorraine Fesq, Stanislav Funiak, Melvin Henry, I-hsiang Shu, Jonathan Kennell, Phil Kim, Raj Krishnan, Michael Pekala, Robert Ragno, John Stedl, John Van Eepoel, Aisha Walcott, David Watson, Andreas Wehowsky, and Margaret Yoon. In addition, we greatly appreciate the early efforts by the NASA LIVINGSTONE group—Pandu Nayak, Bill Millar, Will Taylor, and Jim Kurien.

This research was supported in part by the Defense Advanced Research Projects Agency MOBIES program under contract F33615-00-C-1702, NASA's Cross Enterprise Technology Development Program under contract NAG2-1466, NASA's Intelligent Systems Program under contract NAG2-1388, the Office of Naval Research under contract N00014-99-1-1080, and the DARPA MICA program under contract N66001-01-C-8075.

### References

- Bernard, D.; Dorais, G.; Gamble, E.; Kanefsky, B.; Kurien, J.; Man, G.; Millar, W.; Muscettola, N.; Nayak, P.; Rajan, K.; Rouquette, N.; Smith, B.; Taylor, W.; and Tung, Y. 1999. Spacecraft Autonomy Flight Experience: The DS1 Remote Agent Experiment. Paper presented at the Thirty-Fifth American Institute of Aeronautics and Astronautics Joint Propulsion Conference, 20–24 June, Los Angeles, California.
- Berry, G. 1989. Real-Time Programming: General-Purpose or Special-Purpose Languages. In *Information Processing '89*, ed. G. Ritter, 11–17. New York: Elsevier.
- Davis, R. 1984. Diagnostic Reasoning Based on Structure and Behavior. *Artificial Intelligence* 24(3): 347–410.
- de Kleer, J., and Williams, B. C. 1989. Diagnosis with Behavioral Modes. In Proceedings of the Eleventh International Joint Conference on Artificial Intelligence (IJCAI-89), 1324–1330. Menlo Park, Calif.: International Joint Conferences on Artificial Intelligence.
- de Kleer, J., and Williams, B. C. 1987. Diagnosing Multiple Faults. *Artificial Intelligence* 32(1): 97–130.
- de Kleer, J.; Mackworth, A.; and Reiter, R. 1992. Characterizing Diagnoses and Systems. *Artificial Intelligence* 56(2–3): 197–222.
- Firby, R. J. 1995. The RAP Language Manual, Animate Agent Project Working Note, AAP-6, University of Chicago.
- Harel, D. 1987. Statecharts: A Visual Approach to Complex Systems. In *Science of Computer Programming, Volume 8*, 231–274. Amsterdam, The Netherlands: Elsevier North-Holland.
- Henzinger, T. 1996. The Theory of Hybrid Automata. Paper presented at the Eleventh IEEE Symposium on Logic in Computer Science (LICS '96), 27–30 July, New Brunswick, New Jersey.
- Hofbaur, M., and Williams, B. C. 2002. Mode Estimation of Probabilistic Hybrid Systems. Paper presented at the Fifth International Workshop, Hybrid Systems: Computation and Control (HSCC2002), 25–27 March, Stanford, California.
- Kim, P.; Williams, B. C.; and Abramson, M. 2001. Executing Reactive, Model-Based Programs through Graph-Based Temporal Planning. In Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence, 487–493. Menlo Park, Calif.: International Joint Conferences on Artificial Intelligence.
- Li, S., and Bar-Shalom, Y. 1996. Multiple-Model Estimation with Variable Structure. *IEEE Transactions on Automatic Control* 41(4): 478–493.
- Maybeck, P., and Stevens, R. 1991. Reconfigurable Flight Control via Multiple Model Adaptive Control Methods. *IEEE Transactions on Aerospace and Electronic Systems* 27(3): 470–480.
- Tsamardinos, I.; Muscettola, N.; and Morris, P. 1998. Fast Transformation of Temporal Plans for Efficient Execution. In Proceedings of the Fifteenth National Conference on Artificial Intelligence, 254–261. Menlo Park, Calif.: American Association for Artificial Intelligence.
- Williams, B. C., and Nayak, P. 1997. A Reactive Planner for a Model-Based Executive. In Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence (IJCAI-97), 1178–1185. Menlo Park, Calif.: International Joint Conferences on Artificial Intelligence.
- Williams, B. C., and Nayak, P. 1996a. A Model-Based Approach to Reactive Self-Configuring Systems. In Proceedings of the Thirteenth National Conference on Artificial Intelligence, 971–978. Menlo Park, Calif.: American Association for Artificial Intelligence.
- Williams, B. C., and Nayak, P. P. 1996b. Immobile Ro-

bots: AI in the New Millennium. *AI Magazine* 17(3): 16–35.

Williams, B. C., and Ragno, R. 2004. CONFLICT-DIRECTED A\* and Its Role in Model-Based Embedded Systems. *Journal of Discrete Applied Mathematics* (Special issue on Theory and Applications of Satisfiability Testing). Forthcoming.

Williams, B. C.; Chung, S.; and Gupta, V. 2001. Mode Estimation of Model-Based Programs: Monitoring Systems with Complex Behavior. In Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence, 579–585. Menlo Park, Calif.: International Joint Conferences on Artificial Intelligence.

Williams, B. C.; Ingham, M.; Chung, S.; and Elliott, P. 2003. Model-Based Programming of Intelligent Embedded Systems and Robotic Explorers. *IEEE Proceedings* (Special issue on Modeling and Design of Embedded Software) 9(1): 212–237.

Williams, B. C.; Kim, P.; Hofbaur, M.; How, J.; Kennell, J.; Loy, J.; Ragno, R.; Stedl, J.; and Walcott, A. 2001. Model-Based Reactive Programming of Cooperative Vehicles for Mars Exploration. Paper presented at the 2001 International Symposium on Artificial Intelligence, Robotics, and Automation in Space, 18–22 June, Montreal, Canada.

Young, T.; Arnold, J.; Brackey, T.; Carr, M.; Dwyer, D.; Fogelman, R.; Jacobsen, R.; Kottler, H.; Lyman, P.; Maguire, J.; Pattishall, R.; Soderblom, L.; Staudhammer, P.; Thornton, K.; Wilhelm, P.; Williams, B.; and Zuber, M. 2000. Mars Program Independent Assessment, Technical report to the NASA Administrator and Congress, National Aeronautics and Space Administration, Washington, D.C.



**Brian Williams** received his S.B., S.M., and Ph.D. (1989) from the Massachusetts Institute of Technology (MIT) in computer science and AI. He pioneered multiple-fault, model-based diagnosis in the 1980s at the Xerox Palo Alto Research Center and model-based autonomy in the 1990s through the LIVINGSTONE model-based health management and the BURTON model-based execution systems. 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. On joining the MIT faculty in 1999, 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. He is currently a member of the Advisory Council of the NASA Jet Propulsion Laboratory at the California Institute of Technology. His e-mail address is williams@mit.edu.



**Michael Hofbaur** received a Dipl.-Ing. and a Dr.techn. in electrical engineering from Graz University of Technology, Graz, Austria, in 1995 and 1999, respectively. Since 1999, he has been with the Department of Automatic Control, Graz University of Technology. Since 1999, he has been with the Department of Automatic Control, Graz University of Technology. From 2000 to 2001, he was with the Artificial Intelligence Laboratory at the Massachusetts Institute of Technology as a visiting assistant professor. His research interests include nonlinear and robust control, hybrid systems, automation, and AI. His e-mail address is hofbaur@irt.tugraz.ac.at.



**Gregory Sullivan** is a research scientist at the Massachusetts Institute of Technology (MIT) Artificial Intelligence Laboratory. He received his Ph.D. in 1997 from Northeastern University, where his research focused on formalizing and proving correct compiler optimizations in advanced programming languages. Before coming to MIT, Sullivan worked on developing the dynamic object-oriented language DYLAN. While at MIT, Sullivan has co-founded the Dynamic Languages Research Group; initiated the series of lightweight languages conferences; and conducted and published research on partial evaluation, design patterns, aspect-oriented programming, and dynamic optimization. Sullivan currently focuses on applying advanced design, analysis, and implementation techniques to model-based programming languages. His e-mail address is gregs@ai.mit.edu.



**Michel D. Ingham** is a software engineer and member of the Senior Technical Staff at the Caltech Jet Propulsion Laboratory (JPL). He received his Sc.D. (2003) and S.M. (1998) degrees from the Massachusetts Institute of Technology (MIT) Department of Aeronautics and Astronautics and his B.Eng (1995) in honors mechanical engineering from McGill University. At JPL, Ingham is a member of the software architecture team for the mission data system, where his roles include the de-

velopment of the state analysis modeling and design methodology and the infusion of model-based programming and execution technology into the Mars Science Laboratory and other missions. As a research assistant in the MIT Space Systems Laboratory from 1999 to 2003, he developed the timed model-based-programming paradigm, a novel approach for encoding and robustly executing mission-critical spacecraft sequences and codeveloped the TITAN model-based executive for sequencing and control of robotic spacecraft. His e-mail address is michel.ingham@jpl.nasa.gov.



**Seung Chung** is a member of the Massachusetts Institute of Technology (MIT) Space Systems and Artificial Intelligence Laboratories, working with Brian Williams. He received his Master of Science at MIT in 2003 and was awarded a National Aeronautics and Space Administration Graduate Student Research Program fellowship for his doctoral studies. His current research includes the development of TITAN, a model-based executive capable of autonomously estimating the state of spacecraft, diagnosing and repairing faults, and executing commands. In 2002, he worked with the Artificial Intelligence Group at the Jet Propulsion Laboratory on the development of a distributed model-based executive. His e-mail address is chung@mit.edu.



**Paul Elliott** received an S.B. in aeronautical and astronautical engineering and an S.B. in electrical engineering and computer science from the Massachusetts Institute of Technology in 2001. He is continuing his education there, currently working toward an S.M. in aeronautical and astronautical engineering, and intends to stay for a PhD. He is a research assistant working with Brian Williams. His interests include graphics, software engines, simulators, and compilation. His e-mail address is pelliott@mit.edu.



# The Sixteenth Innovative Applications of Artificial Intelligence Conference

July 27-29, 2004

San Jose Convention Center, San Jose, California  
*Sponsored by the American Association for Artificial Intelligence*

***Call for Papers:***

*[www.aaai.org/Conferences/IAAI/2004/](http://www.aaai.org/Conferences/IAAI/2004/)*

***Paper Submission Deadline:***

*January 20, 2004*