To Appear in AI Magazine, Fall 2003

Model-based Programming of Fault-Aware Systems

Brian C. Williams, Mitch Ingham, Seung Chung, Paul Elliott, and Michael Hofbaur
Space Systems and Artificial Intelligence Laboratories
Massachusetts Institute of Technology, Rm 37-381
Cambridge, Massachusetts 02139
williams,ingham,pelliott,chung,hofbaur@mit.edu

Abstract

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 that 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, while 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 NASA's Deep Space One probe. Model-based programming is being generalized to hybrid discrete/continuous systems and to the coordination of networks of robotic vehicles.

1 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 fly-bys to micro-rovers that can alight upon several asteroids, collect the most interesting geological 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 [1]. 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 US's critical embedded infrastructure on open networks, including the phone system, the internet, and power networks. Opening these systems exposes society to new levels of vulnerability to natural and man-made disasters. 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 or intrusion in any of their constituents. This level of vigilance is common to space missions, which have repeatedly succeeded despite a multitude of hardware failures by employing 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* [2] (Section 2). 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

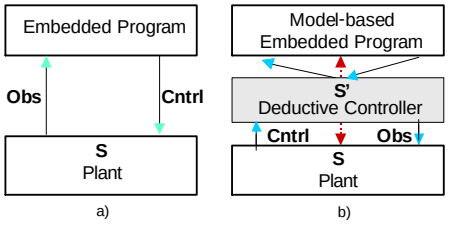

Figure 1: Model of interaction for a) traditional embedded programs and b) model-based programs.

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 (Section 4). 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 (Section 5). In addition, we briefly highlight two other model-based executives: Moriarty, which is used to monitor, diagnose, and control hybrid discrete/continuous systems (Section 7) and Kirk, which coordinates networks of robotic vehicles (Section 8).

2 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 emphasizes executable specifications within the Esterel embedded language: "What you prove is what you execute" [3]. In model-based programming, we carry this 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. This provides the engineer with a simple abstraction that ignores issues of controllability and observability. However, executable specification languages, such as Esterel and Statecharts [4], 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 an error-prone process.

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*, 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*, i.e., states that are not directly observable or controllable. It is then the responsibility of the language's execution kernel to map between hidden states and the plant sensors and control variables. This mapping is performed automatically by employing a deductive controller that reasons from a common-sense 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.

3 Model-based Programming

A model-based program is comprised of two components. The first is a *control program*, which uses standard programming constructs to codify specifications of desired system behavior. In addition, to execute the control program, the execution kernel needs a model of the system it must control. Hence, the second component is a *plant model*, which includes models of the plant's nominal behavior and common failure modes. This modeling formalism, 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, in order to avoid plume contamination. When both are accomplished, thrust one of the two engines, using the other as backup in case of primary engine failure.

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

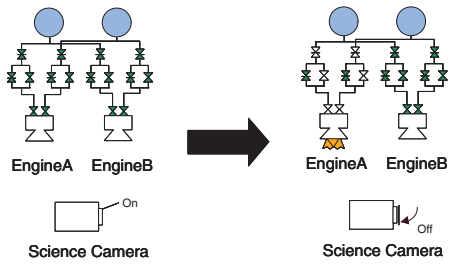

Figure 2: Simple spacecraft for orbital insertion.

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

Figure 3: RMPL control program for orbital insertion.

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

3.1 Control Program

The RMPL control program (Figure 3) codifies the informal specification we gave above 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 by shutting off the camera. This is performed by lines 3-5, where the comma at the end of each line denotes parallel composition. We then fire an engine, choosing to use Engine A as the primary engine (lines 6-9) and Engine B as a backup, in the event that Engine A fails to fire correctly (lines 10-11). Engine A starts trying to fire as soon as it achieves standby and the camera is off (line 7), but aborts if at any time Engine A is found to be in a failure state (line 9). Engine B starts trying to fire only if Engine A has failed, B is in standby and the camera is off (line 10).

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, that is, 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 the corresponding traditional program, which operates on sensed and controlled variables. The added complexity of the traditional program is due to 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 together 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 type of code is tedious. In this situation, the programmer can inadvertantly 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 touch down. The lander then shut off its engine roughly 40 meters above the surface [5].

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.

3.2 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 [6]. Each component automaton is represented by a set of component modes, a set of constraints defining the behavior within each mode, and a set of probabilistic transitions between modes. Constraints are used to represent co-temporal interactions between state variables and intercommunication 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: *off*, *standby* or *firing*. The behavior within each of these modes is described by a set of constraints on plant variables, namely *thrust* and *power_in*. In Figure 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 con-

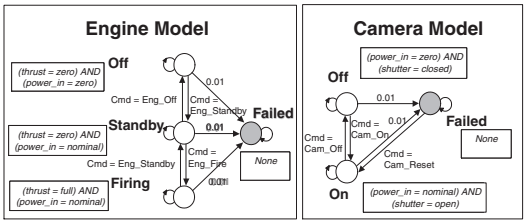

Figure 4: State transition models for a simplified spacecraft.

straints for the engine's behavior in the *failed* mode [7].

A wide range of plant modeling formalisms is possible, depending on the category of system being controlled. These define different families of model-based programming languages. For example, in [8] we represent the plant models as a hybrid between Hidden Markov Models (HMM) and continuous dynamics in order to detect and handle incipient failures. In [9] the plant models are described using a hybrid of hierarchical automata and constraints in order to monitor robotic networks.

4 Model-based Execution of Autonomic 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, 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 goals (*mode reconfiguration*).

Consider a model-based executive, called *Titan*, which coordinates the low-level "autonomic" processes internal to an

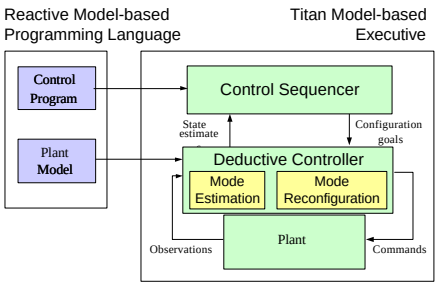

Figure 5: Architecture for a model-based executive.

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

In the process of achieving goal "EngineA = Firing", assume that a failure occurs: an inlet valve to Engine A suddenly sticks closed. Given various sensor measurements (e.g. flow and pressure measurements throughout the propulsion subsystem), the deductive controller identifies the stuck valve as the most likely source of failure. It then tries to execute an alternative control sequence for achieving the configuration goal, for example, by repairing the valve. Presume that the valve 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.

4.1 Mode Estimation

Mode estimation (ME) 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 of Figure 6. Here we assume that ME 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." ME then identifies a number of state transitions that are consistent with this observation, including that either the inlet valve into Engine A has transitioned to stuck closed, as depicted on the right of Figure 6, or that any combination of valves along the fuel or oxidizer supply path are broken.

We frame ME as an instance of belief state update for a HMM. It incrementally 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:

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

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

where $P_{\mathbb{T}}(s_i \mid s_j, \mu^{(t)})$ is defined as the probability that the plant transitions from state s_j to state s_i, given control actions $\mu^{(t)}$, and $P_{\mathbb{O}}(o_k \mid s_i)$ is the probability of observing o_k

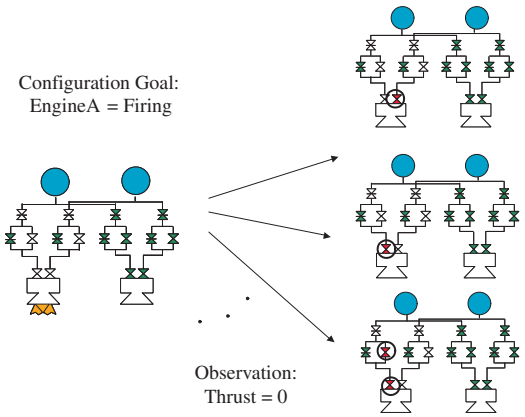

Figure 6: ME step for orbital insertion. Faulty valves are circled and closed valves are filled.

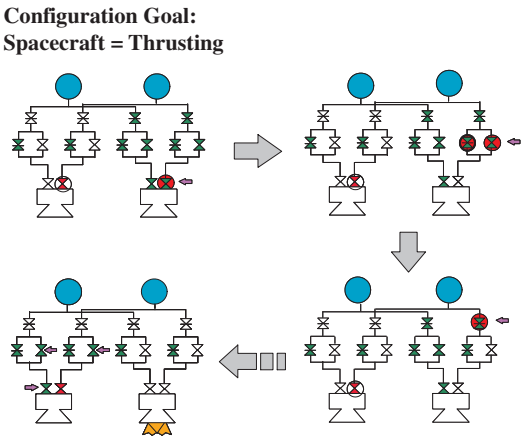

Figure 7: GI uses Conflict-directed A* to search for a mode reconfiguration during orbital insertion.

in state s_i. Probability $p^{(\bullet t+1)}[s_i]$ is conditioned on all observations up to $o^{(t)}$, while $p^{(t+1\bullet)}[s_i]$ is also conditioned on the latest observation $o^{(t+1)} = o_k$.

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, ME enumerates the consistent trajectories and states in order of likelihood using an efficient procedure called *Conflict-directed A**, described below. This offers an any-time approach, which stops enumeration when no additional computational resources are available.

4.2 Mode Reconfiguration

Mode reconfiguration (MR) takes as input a configuration goal $g^{(t)}$, and the most likely current state $s^{(t)}$ computed by ME, and returns a series of commands that progress the plant towards a maximum-reward goal state that achieves $g^{(t)}$ [10]. MR accomplishes this using a *goal interpreter* (GI) and a *reactive planner* (RP). GI determines a target state $s_g^{(t)}$ that is reachable from $s^{(t)}$ and that achieves $g^{(t)}$, while maximizing reward. It accomplishes this by having Conflict-directed A* search over the reachable states in best-first order. RP generates a command sequence that moves the plant from $s^{(t)}$ to $s_g^{(t)}$. RP generates and executes this sequence one command at a time, using ME to confirm the effects of each command.

For example, in our orbital insertion example, given a configuration goal of "EngineB = Firing," GI selects a set of valves to open that establish a flow of fuel to the engine (bottom left, Figure 7). RP sends commands to control units, drivers and valves to achieve this target.

4.3 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 [6], performed MR. However, much of the complexity of MR 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.

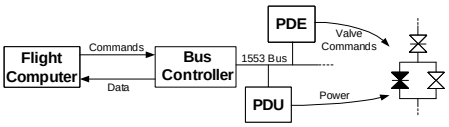

Figure 8: A spacecraft establishes complex interaction paths from the flight computer to the main engine.

The following is an example of a scenario that a robust close-valve routine should be able to handle. The corresponding procedure is automatically generated by Titan's RP:

> 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 will need 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 be quickly repaired. To address these challenges, Titan's RP 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. This is in contrast to explicit universal plans, whose size is exponential in the number of components. Titan's reactive planner, called Burton, is developed in [2; 10].

5 Efficient On-line Reasoning Through Conflict-directed A*

The core problems underlying model-based programming, such as ME and GI, 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 $\mathbf{y}$, each ranging over a finite domain, a utility function f on $\mathbf{y}$, and a set of constraints C that $\mathbf{y}$ must satisfy. A solution is an assignment to $\mathbf{y}$ that maximizes f and satisfies C. For ME, 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 GI, 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.

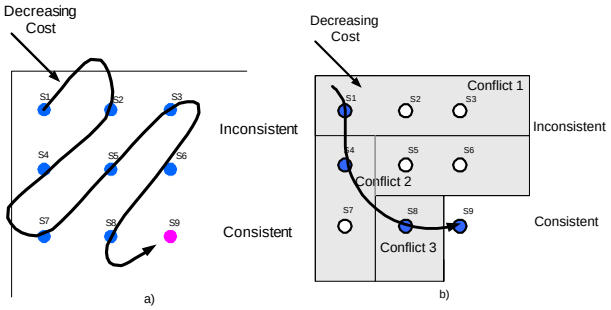

Figure 9: a) A* examines all best cost states up to the solution. b) Conflict-directed A* jumps over best cost states contained by known conflicts.

The Conflict-directed A* method, employed by Titan, also searches in best-first order, but accelerates search by eliminating subspaces around each state that are inconsistent. This process builds upon the concepts of conflict and kernel diagnosis introduced in model-based diagnosis [11; 12]. 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 $S1$ - $S3$. Conflict-directed A* then tests $S4$ as the highest utility state resolving Conflict 1. $S4$ tests inconsistent and generates Conflict 2, eliminating states $S4$ - $S7$. Conflict-directed A* continues until it finds $S9$ consistent, and hence an optimal solution. Figure 7 shows a similar conflict-detection sequence, generated by GI 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. Finally, 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 [13], and makes rigorous a set of focussing mechanisms first introduced heuristically within the Sherlock diagnosis system [14], and evolved within the model-based diagnosis community over the last decade. Though 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.

6 Model-based Execution Six Light Minutes From Earth

Titan and its predecessors, Livingstone[6], Sherlock[14], and GDE[11], 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 MIT SPHERES multi-spacecraft mission, which is on the manifest to be flown *inside* the International Space Station. In addition, we are working in colaboration with the Caltech Jet Propulsion Laboratory to apply RMPL and Titan to NASA's Mars 2009 rover mission. Titan has also been applied to testbeds for the 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 reconfiguration system [6]. 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 [15]. DS1 is an asteroid and comet fly-by mission, which used an ion propulsion system and navigated by the stars using a cam-

era and onboard 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 in order 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, while 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 is the control and fault management of a multi-vehicle cooperative cruise control system, developed at University of California, Berkeley. The second is an onboard automobile fault management system, which is being developed in collaboration with Toyota.

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

7 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 forces 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 without a trace. It most likely crashed into Mars after it incorrectly shutdown its engine 40 meters above the surface, because it misinterpreted its altitude due to a faulty software monitor.

The above case study is a dramatic instance of a common problem – increasingly complex systems are being developed whose failure symptoms are nearly indiscernible, up 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 [8]. Failures may 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.

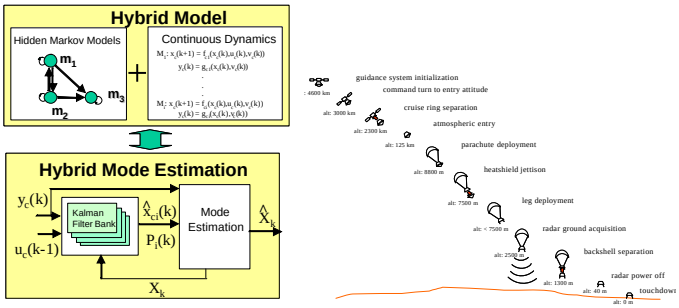

Figure 10: left) Hybrid mode estimation, right) Mars entry, descent and landing sequence (compliments of NASA JPL).

Moriarty extends Titan's concurrent probabilistic constraint automata model to include continuous dynamical system models as constraints (top left, Figure 10). This is unlike most traditional hybrid modeling frameworks (for example, [16]), which 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 estimate a system's mode changes and coordinates the actions of a set of continuous state observers. This approach is similar to work pursued in multi-model estimation [17; 18]. However, Moriarty provides a novel any-time, 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.

8 Model-based Programming of Robotic Networks

Thus far we have focussed on model-based programming methods that increase robustness and autonomy by generating the autonomic 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 may be a heterogenous collection of planes, helicopters, boats and ground vehicles that perform search-and-rescue during natural or man-made disasters, or a set of rovers, blimps and orbiters exploring science sites on Mars (left, 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

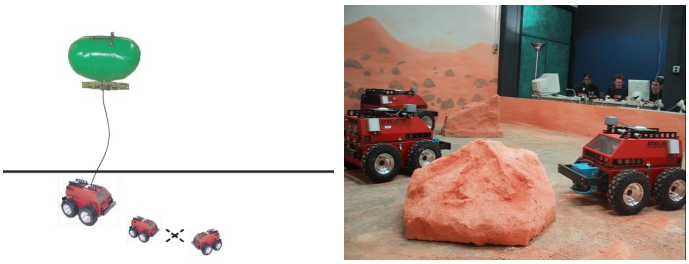

Figure 11: Mars exploration using rovers and blimps.

specifying global strategies for multi-vehicle coordination. We refer to this extended language as *cooperative RMPL* and its corresponding executive as *Kirk* [19]. 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 dynamics of each robot vehicle in the network.

Most embedded programming languages and robotic execution languages, such as [20], employ myopic execution strategies that do not evaluate their future course of action in terms of feasability or optimality. Kirk is distinguished in that it first "looks" by employing 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 [21], which adapts to execution uncertainties through fast, online scheduling.

To "look," Kirk's control sequencer chooses a set of threads of execution from a nondeterministic RMPL program, producing a *configuration plan*, and checks to ensure that this plan is consistent and schedulable. The plan is comprised of 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 deductive controller employ variants of graph-based temporal planning to accelerate reasoning.

The Mars exploration concept has been validated within the MIT cooperative robotics testbed using four ATRV rovers and an overhead stereo camera, emulating a blimp (right, Figure 11) [22]. Kirk has also been demonstrated in simulation on the coordination of up to nine air vehicles performing a suppression of enemy air defense mission.

9 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 or her thinking to the level of specifying intended state evolutions, while 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. An 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 Deep Space One probe in the Spring of 1999. Titan is currently being demonstrated on automobiles, Mars rovers and a micro-spacecraft 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.

10 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, Jon Kennell, Phil Kim, Raj Krishnan, Michael Pekala, Robert Ragno, John Stedl, John Van Eepoel, Aisha Walcott, Dave 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 DARPA 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 ONR under contract N00014-99-1-1080, and the DARPA MICA program under contract N66001-01-C-8075.

References

[1] B. C. Williams, P. P. Nayak, Immobile robots: AI in the new millennium, in: AI Magazine, Vol. 17(3), 1996, pp. 16–35.

[2] B. C. Williams, M. Ingham, S. Chung, P. Elliott, Model-based programming of intelligent embedded systems and robotic explorers, IEEE Proceedings, Special Issue on Embedded Software .

[3] G. Berry, Real-time programming: General purpose or special-purpose languages, in: G. Ritter (Ed.), Information Processing 89, Elsevier, 1989, pp. 11 – 17.

[4] D. Harel, Statecharts: A visual approach to complex systems, Science of Computer Programming 8 (1987) 231 – 274.

[5] T. Young, *et al.*, Mars program independent assessment, Tech. rep., Report to the NASA Administrator and to Congress (March 2000).

[6] B. C. Williams, P. Nayak, A model-based approach to reactive self-configuring systems, in: Proc AAAI-96, 1996, pp. 971–978.

[7] R. Davis, Diagnostic reasoning based on structure and behavior, Artificial Intelligence 24 (1984) 347–410.

[8] M. Hofbaur, B. C. Williams, Mode estimation of probabilistic hybrid systems, in: Intl. Conf. on Hybrid Systems: Computation and Control, 2002.

[9] B. C. Williams, S. Chung, V. Gupta, Mode estimation of model-based programs: Monitoring systems with complex behavior, in: Proceedings of IJCAI-01, 2001.

[10] B. C. Williams, P. Nayak, A reactive planner for a model-based executive, in: Proceedings of IJCAI-97, 1997.

[11] J. de Kleer, B. C. Williams, Diagnosing multiple faults, Artificial Intelligence 32 (1) (1987) 97–130.

[12] J. de Kleer, A. Mackworth, R. Reiter, Characterizing diagnoses and systems, Artificial Intelligence 56 (1992) 197–222.

[13] B. C. Williams, R. Ragno, Conflict-directed a* and its role in model-based embedded systems, J. of Discrete Applied Math, Special Issue on Theory and Applications of Satisfiability Testing .

[14] J. de Kleer, B. C. Williams, Diagnosis with behavioral modes, in: Proc IJCAI-89, 1989, pp. 1324–1330.

[15] D. Bernard, G. Dorais, E. Gamble, B. Kanefsky, J. Kurien, G. Man, W. Millar, N. Muscettola, P. Nayak, K. Rajan, N. Rouquette, B. Smith, W. Taylor, Y. Tung, Spacecraft autonomy flight experience: The DS1 remote agent experiment, in: Proceedings of the AIAA, 1999.

[16] T. Henzinger, The theory of hybrid automata, in: Proceedings of the IEEE Symposium on Logic in Computer Science (LICS '96), 1996, pp. 278–292.

[17] P. Maybeck, R. Stevens, Reconfigurable flight control via multiple model adaptive control methods, IEEE Transactions on Aerospace and Electronic Systems 27 (3) (1991) 470–480.

[18] X. Li, Y. Bar-Shalom, Multiple-model estimation with variable structure, IEEE Transactions on Automatic Control 41 (1996) 478–493.

[19] P. Kim, B. C. Williams, M. Abramson, Executing reactive, model-based programs through graph-based temporal planning, in: Proceedings of IJCAI-01, 2001.

[20] R. J. Firby, The RAP language manual, Animate Agent Project Working Note AAP-6, University of Chicago (March 1995).

[21] I. Tsamardinos, N. Muscettola, P. Morris, Fast transformation of temporal plans for efficient execution, in: Proceedings of AAAI-98, 1998.

[22] B. C. Williams, P. Kim, M. Hofbaur, J. How, J. Kennell, J. Loy, R. Ragno, J. Stedl, A. Walcott, Model-based reactive programming of cooperative vehicles for mars exploration, in: Proceedings of ISAIRAS-01, 2001.