Model-Based Programming of Intelligent Embedded Systems Through Offline Compilation

K. Sundar Srinivas[1], K. Naga Lakshmi Kalyani[2]
[1]M.Tech, Dept. of ECE, Nova College of Engineering & Technology, A.P., India.
[2]Asst.Professor, Dept. of ECE, Potti Sriramulu College of Engineering & Technology, A.P., India.

Abstract - **Many recent and future space missions point to the need for increased autonomy in spacecraft with an emphasis on more capable fault diagnostic systems. The most widely used fault diagnostic systems are rule-based. Rule-based systems have quick response to events and clearly present to engineers the predefined reactions to events. These systems, however, require engineers to manually generate all necessary rules and these do not convey the assumed model the engineers used to generate the rules. Contrarily, model-based systems eliminate the need to manually generate the rules. Programming complex embedded systems involves reasoning through intricate system interactions along lengthy paths between sensors, actuators, and control processors. This is a challenging, time-consuming, and error-prone process requiring significant interaction between engineers and software programmers. Furthermore, the resulting code generally lacks modularity and robustness in the presence of failure. Model-based programming addresses these limitations, allowing engineers to program reactive systems by specifying high-level control strategies and by assembling commonsense models of the system hardware and software.**

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

I. INTRODUCTION

Recent failures in NASA's Mars exploration program point to the need for increased autonomy in spacecraft. Spacecraft must be designed with the capacity to monitor their own systems for unexpected occurrences, and to react in a timely fashion to such conditions at the executive layer, i.e. at the level of real-time commanding. The ability to accurately and rapidly determine the current state of the system is vital to the design of fault protection systems in autonomous spacecraft. Many fault management systems are based on expert systems in which a rule-based diagnostic engine is used to detect faults. For example, the NEAR spacecraft used such a system for limited autonomous operations. This type of system's capability is limited to the rules enumerated in the database. To create these rules, engineers must reason through system wide interactions, consequently, the set of rules is limited by the faults that engineers can recognize. This lack of robustness can be detrimental to the spacecraft. Components may interact other than expected, and a rule-based system cannot account for this. Should such a fault occur at a critical mission point, such as orbital insertion, the rule-based engine cannot react, resulting in the loss of the mission?

Embedded systems, from automobiles to office-building control systems, are achieving unprecedented levels of robustness by dramatically increasing their use of computation.

We envision a future with large networks of highly robust and increasingly autonomous embedded systems. These visions include intelligent highways that reduce congestion, cooperative networks of air vehicles for search and rescue, and fleets of intelligent space probes that autonomously explore the far reaches of the solar system. Many of these systems will need to perform robustly within extremely harsh and uncertain environments, or may need to operate for years with minimal attention. To accomplish this, these embedded systems will need to radically reconfigure themselves in response to failures, and then accommodate these failures during their remaining operational lifetime. We support the rapid development of these systems by creating embedded programming languages that are able to reason about and control underlying hardware from engineering models. We call this approach *model-based programming*.

A. Robustness in Deep Space

In the past, high levels of robustness under extreme uncertainty were largely the realm of deep-space exploration. Billion-dollar space systems, like the Galileo Jupiter probe, have achieved robustness by employing sizable software development teams and by using many operations personnel to handle unforeseen circumstances as they arise. Efforts to make these missions highly capable at dramatically reduced costs have proven extremely challenging, producing notable losses, such as the Mars Polar Lander and Mars Climate Orbiter failures [1]. A contributor to these failures was the inability of the small software team to think through the large space of potential interactions between the embedded software and its underlying hardware.

For example, consider the leading hypothesis for the cause of the Mars Polar Lander failure. Mars Polar Lander used a set of Hall Effect sensors in its legs to detect touchdown. These sensors were watched by a set of software monitors, which were designed to turn off the engine when triggered. As the lander descended into the Mars atmosphere, it deployed its legs. At this point it is most likely that the force of deployment produced a noise spike on the leg sensors, which was latched by the software monitors. The lander continued to descend, using a laser altimeter to detect distance to the surface. At an altitude of approximately 40 m, the lander began polling its leg monitors to determine touchdown. It would have immediately read the latched noise spike and shut down its engine prematurely, resulting in the spacecraft plummeting to the surface from 40 m [2].

B. A MODEL-BASED PROGRAMMING EXAMPLE

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

Heat up both engines (called standby mode). Meanwhile, turn the camera off, in order to avoid plume contamination

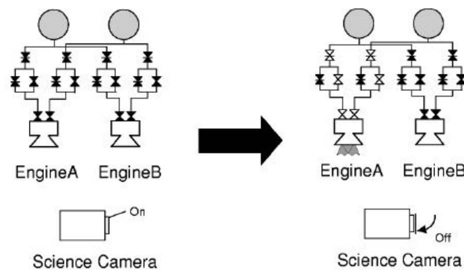

Fig. 3 Simple spacecraft for the orbital insertion scenario. Initial state (left) and goal state (right) are depicted.

When both are accomplished, thrust one of the two engines, using the other engine as backup in case of primary engine failure.

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

2. MINI-ME

Mini-ME differs from previous model-based fault monitoring systems by guaranteeing run time performance. Through model compilation and offline deduction, Mini-ME combines the benefits of the rule based system's real time performance guarantees and the model-based fault protection system's capability to reason on models.

2.1 Example System

The diagnostic ability of Mini-ME will be demonstrated in the following sections using a simplified schematic of a monopropellant propulsion system used for attitude control in the NEAR spacecraft, shown in Figure 2.

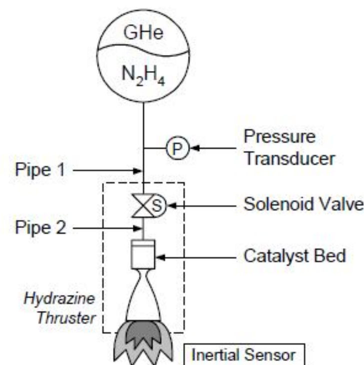

Figure 2: Monopropellant Propulsion System Schematic

The propulsion system comprises two overall subsystems, the tank of hydrazine and its associated pressure transducer, and the hydrazine thruster made up of the solenoid valve, catalyst bed and physical thruster. An inertial sensor is included in the system for thrust observation.

The hydrazine thruster is made up of two main components, the solenoid valve and the catalyst bed. The solenoid valve controls the hydrazine flow into the catalyst bed. This is accomplished by applying an electric current to the valve to open it, otherwise it will remain closed. Downstream of the solenoid valve is the catalyst bed, which is needed for combustion. Over time, catalyst can be lost through various mechanisms, such as pieces breaking off due to temperature variations. This will cause a reduction in thrust from the hydrazine thruster, causing the inertial sensor to observe that the thrust is off. In the case

that the ACS operates nominally, the inertial sensor's reading will be discretized as on.

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

A. Control Program

The RMPL control program, shown in Fig. 4, encodes the informal specification we gave previously as a set of state trajectories. The specific RMPL constructs used in the program are introduced in Section IV. Recall that to perform orbital insertion, one of the two engines must be fired. We start by concurrently placing the two engines in the standby state and by shutting off the camera. This is performed by lines 3–5, where commas at the end of each line denote parallel composition. We then fire an engine, choosing to use Engine A as the primary engine (lines 6–9) and Engine B as a backup, in the event that Engine A fails to fire correctly (lines 10–11). Engine A starts trying to fire as soon as it achieves standby and the camera is off (line 7), but aborts if at any time Engine A is found to be in a failure state (line 9). Engine B starts trying to fire only if Engine A has failed, B is in standby, and the camera is off (line 10). Several features of this control program reinforce our earlier points. First, the program is stated in terms of state assignments to the engines and camera, such as "EngineB = Firing." Second, these state assignments appear both as assertions and as execution conditions. For example, in lines 6–9, "EngineA = Firing" appears in an assertion (line 8), while "EngineA = Standby," "Camera = Off," and "EngineA = Failed" appear in execution conditions (lines 7 and 9). Third, none of these state assignments are directly observable or controllable, only shutter position and acceleration may be directly sensed, and only the flight computer command may be directly set. Finally, by referring to hidden states directly, the RMPL program is far simpler than a corresponding program that operates on sensed and controlled variables. The added complexity of the latter program is due to the need to fuse sensor information and generate command sequences under a large space of possible operation and fault scenarios.

B. Plant Model

The plant model is used by a model-based executive to map queried and asserted states in the control program to sensed variables and control sequences, respectively, in the physical plant. The plant model is built from a set of component models. Each component is represented by a set of component modes, a set of constraints defining the behavior within each mode, and a set of probabilistic transitions between modes. The component automata operate concurrently and synchronously.

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

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

III. RULE SYSTEM ANALYSIS

A comparison to a real system is the best validation for the Mini-ME fault diagnosis tool. For verification, a NEAR-like power system and its associated rules were analyzed to develop appropriate Mini-ME models to obtain diagnoses of particular faults. These rules have several characteristics relating to Mini-ME, the first being the dependency on time. In all of the rules, the observation must be made for a certain length of time before it is triggered. This dependency is moved outside of Mini-ME through the use of the monitors. Monitors can be designed with a counter that is incremented when an observation falls in a certain range, such as if the charger current exceeds 0.8 A. Only when the counter reaches a certain value, corresponding to 10 seconds for rule 3, then would the monitor send the observation that the charger current is "high". This use of discretization allows the modeling to be more intuitive and understandable as the model is now specified in a more qualitative way.

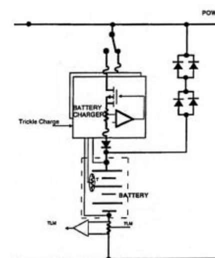

Figure 4 - NEAR-like Power Storage Sub-system

Table 2. NEAR-like Power Storage System Rules

No.	Symptom	Recovery Action
1	(Battery Current > 0.6A) For 60 sec	Turn off the charger
2	(Redundant battery charger is ON) for 5 sec	stop rules 2 and 3
3	(Charger current > 0.8 A) for 10 sec	Switch to the redundant charger, and disengage the primary.
4	(Charger current > 0.07A) and (Bus Voltage > 24 V) for 10 sec	Same recovery as rule 2.
5	(Battery Temp > 30 C) for 1 hour	Switch to the redundant charger and turn its trickle charge on

A second characteristic of a NEAR-like rule is that different symptoms can lead to the same recovery action, such as the conditions for switching to the redundant charger (rules 3 and 4). These types of rule combinations may have the same resulting action, but lead to a different state for the component. Hidden in these rules then is the state of the power system that the engineer had to determine. For instance, in the case of rule 3, this would mean that the charger has become broken in some way, thus identifying the state, and the model of this mode would come from these symptoms.

Mini-ME has also been used as a tool to understand model compilation techniques and its applications. Other such applications that use this technique include a mode estimation capability for the Reactive Model based Programming Language (RMPL) [3], and a reactive planning system, Burton [4].

IV. CONCLUSION

Fault protection in spacecraft is a must as missions venture further into space and space systems increase in complexity. The necessity of a system that can perform this fault diagnosis in real time is then a key component. The Mini-ME fault protection system has been shown to meet this goal without any loss of information from a rule-based system. The utilization of system models in Mini-ME allows it to perform diagnosis of components. A model-based approach has many benefits including reusability, compositionality and specification of intuitive models. The use of these models to perform reasoning and deduction has been shifted to an offline operation, an approach that differs from previous systems such as Sherlock and Livingstone. This offline compilation of the models to rules, called dissents, allows Mini-ME to perform fast diagnosis of faults online. Using these models and observations from the system, Mini-ME generates a diagnosis of the system's components using a best first search to generate the most likely diagnosis. This diagnosis gives the state of the system, which is not available in a rule-based system. In rule-based systems, the mapping from symptoms to recovery action is apparent, but not the mapping from symptoms to the system state. Making this step explicit leads to rules that are easier to analyze for completeness, and a rule set smaller in size. In the case of the example system in section 2, it requires only 6 dissents to represent the faults, whereas a rule-based system would require 32 rules to represent all of the possible faults. These characteristics lead to more reliable fault protection as it makes the process of rule generation modular by using models of the system, monitors that discretized observations and repair actions based on the diagnoses, all of which are designed by the engineer in a clear manner. A key benefit of the Mini-ME system and the use of associated repair manager, aside from the model-based approach, is that they give the spacecraft the ability to remain operational in the face of component failures. This ability is crucial as space exploration expands. The same individuals who designed the spacecraft may not be around when it lands, which necessitates fault diagnosis ability.

Acknowledgment

I am thankful to my Guide **Miss. *K. Naga Lakshmi Kalyani*** Asst.Professor Dept.of ECE for her valuable suggestion to complete my thesis paper in intime.

REFERENCES

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

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

[3] B. Williams and P. Nayak. 1997. "A Reactive Planner for a Model-based Executive." In Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI-97).

[4] B. Williams, S. Chung, and V. Gupta. Mode Estimation of Model-based Programs: "Monitoring Systems with Complex Behavior," To appear in *Proceedings of the International Joint Conference on Artificial Intelligence*, Seattle,WA.2001.around when it lands, which necessitates fault diagnosis ability.