Practical Application of Model-based Programming and State-based Architecture to Space Missions

Gregory A Horvath, Michel D Ingham
Jet Propulsion Laboratory
California Institute of Technology
4800 Oak Grove Drive
Pasadena, CA 91109
{gregory.horvath, michel.ingham}@jpl.nasa.gov

Seung Chung, Oliver Martin, Brian Williams
CSAIL
Massachusetts Institute of Technology
77 Massachusetts Avenue
Cambridge, MA 02139
{chung, omartin, williams}@mit.edu

Abstract

Innovative systems and software engineering solutions are required to meet the increasingly challenging demands of deep-space robotic missions. While recent advances in the development of an integrated systems and software engineering approach have begun to address some of these issues, they are still at the core highly manual and, therefore, error-prone. This paper describes a task aimed at infusing MIT's model-based executive, Titan, into JPL's Mission Data System (MDS), a unified state-based architecture, systems engineering process, and supporting software framework. Results of the task are presented, including a discussion of the benefits and challenges associated with integrating mature model-based programming techniques and technologies into a rigorously-defined domain specific architecture.

1 Introduction

1.1 Motivation and Objectives

As spacecraft and their missions have become ever more complex and ambitious, systems and software engineering practice has been severely challenged to create and verify space systems that assure correctness, reliability, and robustness in an efficient, cost effective manner. These challenges have motivated the development of the Mission Data System (MDS), a unified multi-mission software architecture for flight, ground, and test systems, and a novel systems engineering methodology, called State Analysis. State Analysis produces model-based requirements on the system and software design, which map explicitly to components in the MDS software product line, an adaptable set of software frameworks for developing embedded control systems. This explicit mapping narrows the gap between systems and software engineering, and reduces the risk of introducing errors during software implementation. However, the models and software specifications from State Analysis still require manual translation into robust software for estimation and control. This translation process implies that the possibility of errors still exists.

The objective of this task is to augment State Analysis and MDS with the benefits of Model-based Programming, an approach to engineering software that is able to directly reason through models of a system for the purposes of estimation, control, fault diagnosis, recovery, planning, and execution. The Model-based Programming paradigm is embodied in a suite of software technologies developed at MIT, including the Titan model-based executive. In the Titan system, models of component behavior are fed directly into a reasoning engine in order to compute estimates of the current system state, and deduce commands necessary to drive the system towards a specified goal state. This paper will describe the integration of Titan into the MDS framework and, more generally, will address issues associated with integrating advanced Model-based Programming technology into flight and ground software for space exploration missions.

1.2 Outline

The paper begins with a background section, briefly exploring the basics of the two fundamental technologies involved in the task, the MDS architecture (including the State Analysis process) and Model-based Programming. A detailed discussion of the task follows, including a description of the motivations for the task, the task objectives, and the technical approach. The paper concludes with a discussion of results and a breakdown of challenges addressed by the task.

2 Background

2.1 MDS

MDS is a state-based, unified multi-mission architecture for creating integrated flight, ground, and simulation systems. It is supported by the State Analysis process, a rigorous systems engineering methodology. State Analysis produces a set of requirements on system and software design based on models of the system state behavior. These model-based requirements map directly to a software implementation thanks to frameworks in which the main elements of State Analysis are explicitly represented as software components. As a result, requirements are more formal than textual 'shall' statements, the translation from requirements to code is less ambiguous, and software and systems engineers communicate using a common language, reducing the chance for errors introduced by miscommunication between the two domains.

2.1.1 State-based Control Architecture

The state-based control architecture, depicted in Figure 1, is designed to meet the needs of control systems for complex embedded applications. It is built on several overarching principles which help ensure consistency throughout the entire product lifecycle [1, 2], including the following:

- Models are central to the entire system. Models of system behavior are documented and used to inform all aspects of system development and design.
- Requirements developed according to the State Analysis process map directly into a software implementation. The MDS software frameworks, discussed below, are one implementation of a component-centric state-based framework.
- All knowledge of system state is stored in *State Variables*. Thus, all components that require information about state get it from a common source, eliminating errors which arise as a result of disparate components using private data to reason about the state of the system.
- Estimation and control, which are often intertwined in traditional flight and ground software architectures, are strictly separated. This makes the logic for each function explicit, as well as easier to review. Estimators and controllers are also referred to as *achievers*, since their role is to achieve the goals issued by the Mission Planning and Execution software.
- Operator intent is made explicit through the use of *goals*, which are constraints on the value of a given state variable over a time interval, rather than inferred via examination of low-level command sequences.

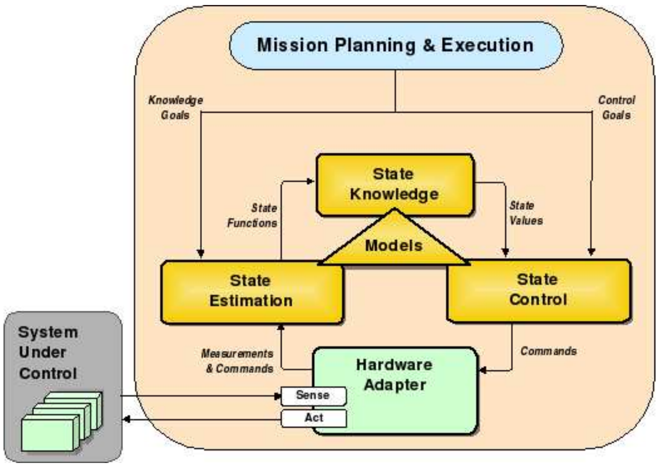


Figure 1. State-Based Control Architecture [1]

2.1.2 State Analysis

State Analysis is a rigorous model-based systems engineering methodology that leverages the state-based control architecture [1]. State Analysis aims to improve on the current state of systems engineering practice by documenting requirements of the target system in terms of models, not textual 'shall' statements. The more formal and thorough requirements specification helps to reduce the ambiguity inherent in the traditional requirements capture process, and provide systems and software engineers with a common language to communicate. When coupled with a state-based software framework like MDS, the specifications produced through State Analysis map directly to code.

Just as in the architecture, state is central to the process. Systems engineers begin the process with a State Discovery and Modeling activity, in which the key state variables, commands and measurements for the system are identified in an iterative fashion, and models of the behavior of these state variables are incrementally constructed. Our model of the system under control is composed of:

- *State Effects Models* describing how each state in the system under control evolves over time and under the influence of other states;
- *Measurement Models* describing how each measurement is affected by various states in the system under control; and
- *Command Models* describing how states are affected by each command (possibly under the influence of other states in the system under control).

In addition to these models, the State Discovery and Modeling activity produces *State Effects Diagrams* (SEDs), which depict state variables as ovals, measurements as upward-facing triangles, and commands as downward-facing triangles (see Figure 2). An arrow from one state

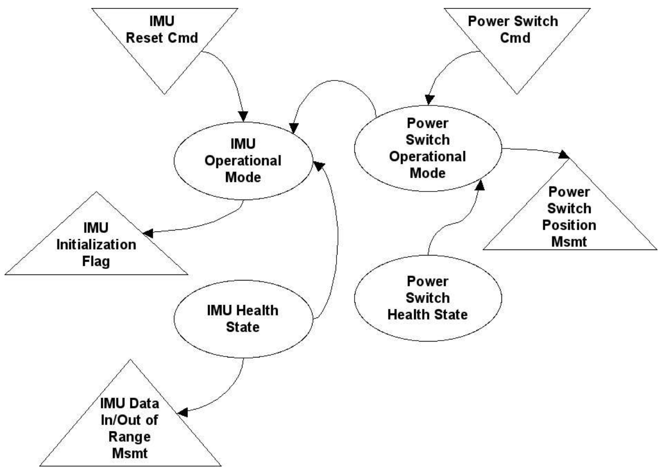

IMU Reset Cmd
Power Switch Cmd
IMU Operational Mode
Power Switch Operational Mode
Power Switch Position Msmt
IMU Initialization Flag
IMU Health State
Power Switch Health State
IMU Data In/Out of Range Msmt

Figure 2. IMU Adaptation State Effects Diagram

variable to another indicates a causal (physical) effect between these state variables. An arrow from a state variable to a measurement indicates that the measurement is affected by that state variable. An arrow from a command to a state variable indicates that the command has an effect on that state variable. The state effects diagrams and models are essential products of the State Analysis process, and are used to inform many tasks performed later in the process, such as estimator and controller design, software collaboration diagramming, and specification of goals and activity elaborations.

2.1.3 MDS Frameworks

MDS also incorporates a robust set of software frameworks, whose structure closely mirrors that of the state-based architecture. As a result, the mapping from requirements (developed through State Analysis) to software implementation is straightforward.

The MDS software frameworks are a fully-featured set of reusable classes that developers can use to leverage the benefits of State Analysis and the state-based architecture. The frameworks are structured in a layered fashion, and provide facilities from the OS abstraction layer all the way up through the application layer. The frameworks contain many basic services, such as math packages, serialization facilities, and an efficient type ID system, as well as higher-level components that map to the state-based architecture, such as state variables, estimators and controllers, and planning and execution functionality. The MDS software frameworks are currently available in C++ and Java implementations. The C++ version was used in support of this task.

The software design specifications captured as part of the State Analysis process are readily mapped to canonical patterns of modular and reusable software components in the MDS software frameworks. These software components are then implemented as *adaptations* of the MDS software framework, following a set of well-documented steps.

By encompassing all aspects of the development process, from architecture to systems engineering to software implementation to operations, MDS provides an environment for creating highly reliable software products throughout the entire project lifecycle.

2.2 Model-Based Programming and the Titan Model-Based Executive

As control programs for embedded systems become more complex, the job of programmers becomes more complex: they must reason about interactions between many components, as well as inferring system state from an incomplete set of sensor data. As the complexity of components and the number of interactions between components increase, the traditional approach to developing control software becomes unmanageable. Model-based executives address this problem by reasoning about declarative models of the system in order to inform the processes of estimation, control, activity planning, and robust activity execution [3].

The benefits of the model-based programming approach are numerous, and applicable at each stage of the development process. In the requirements capture phase, the use of formally specified declarative models instead of the typical shall-statements leads to more detailed, inspectable and reviewable specifications. During the design phase, the coverage of estimation and control software is broadened: by focusing on model development and allowing the executive to manage the complex inter-component interactions, monitoring and control functionality can be elevated to the subsystem or system level. This allows, for example, coverage of dual fault cases (which are usually too complex for consideration in typical space missions). During implementation, the difficult and error-prone task of translating requirements into code is eliminated, since the model is the input to the control software. Finally, the formal behavioral specifications can be used to perform model checking during verification and validation (V&V).

Model-based programming is an emerging technology that has made considerable impact. At the core, model-based executives must solve an instance of the classic AI planning problem. Combined with the memory and processing constraints presented by highly resource-constrained embedded platforms, performance is a paramount concern. However, recent advances have led to an efficient and accurate model-based executive, and its performance continues to improve with ongoing research. For example, performance results presented in [4] for the latest generation of model-based reasoning algorithms show a significant improvement over the Livingstone algorithm that was flight validated as part of the Remote Agent Experiment on the Deep Space 1 mission [5].

Other perceived risks associated with Model-based Programming relate to how this approach scales to larger sys-

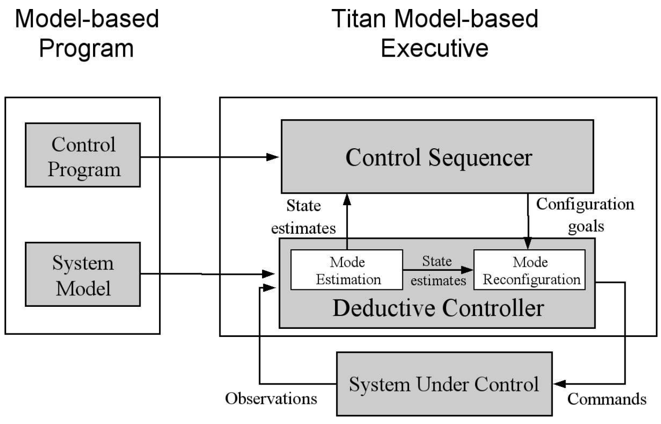


Figure 3. Titan Architecture Diagram [3]

tems, and the implications on the V&V process. Reference [4] begins to address the scalability issue, and reference [6] identifies the key issues associated with V&V of model-based software systems and ways to address them. Overall, the field of Model-based Programming promises to bring many advances to embedded software development in order to meet the needs of future applications.

Titan is an implementation of a model-based, system-level autonomy framework, developed at MIT. The Titan architecture, shown in Figure 3, consists of two main components: a control sequencer and a deductive controller. The deductive controller takes as input a model of the system, observations of the system, and any commands that have been issued to generate state estimates; this process is termed *Mode Estimation*. The control sequencer uses these state estimates and a control program written in RMPL (Reactive Model-based Programming Language), to generate *configuration goals*. Each configuration goal is provided to the deductive controller, which uses the state estimate to determine what actions are necessary to drive the system toward a least-cost goal state that satisfies the configuration goal; this process is called *Mode Reconfiguration*. In MDS, the Mission Planning and Execution (MPE) software performs the function of the control sequencer; as such, this task focused on leveraging the functionality of Titan's deductive controller. Accordingly, the proceeding discussion will be limited to the capabilities provided by the deductive controller.

2.2.1 Mode Estimation

Mode Estimation (ME) is the process of determining the current best estimate(s) of system state. The deductive controller takes as input observations of the system state, any commands that may have been issued, as well as a model of the system of interest, expressed as a factored Partially-Observable Markov Decision Process (POMDP) [3]. It generates as output a set of system state estimates; each system state estimate consists of an assignment to each state variable and an associated probability [7]. The ME process is repeated at each time step, with the set of K most likely system states serving as the starting point of the next estimation cycle.

Early versions of Titan's deductive controller employed Livingstone's Best-First Trajectory Enumeration (BFTE) method for estimating system state, in which estimates are generated by expanding a tree of possible states at each step [7]. The BFTE algorithm was a significant advance in the development of a practical model-based executive; however, a new algorithm has been developed which improves the performance and accuracy of the ME process and, consequently, the overall performance of the deductive controller. The new ME algorithm uses a process called Best-First Belief State Enumeration (BFBSE), which efficiently computes an approximate belief state, adding probabilities associated with different possible transitions to the same end state. Initial empirical and analytical data show a significant performance improvement over BFTE in terms of memory, time, and accuracy [7].

The version of Titan used in support of this task is based on an implementation of the Best-First Belief State Update (BFBSU) algorithm, which further improves on the accuracy of the BFBSE algorithm by reducing the number of false-positive diagnoses of fault modes with several consistent observations [8].

2.2.2 Mode Reconfiguration

Mode Reconfiguration (MR) is the process of determining which (if any) commands must be issued in order to drive the system towards a specified state that satisfies a *configuration goal*, issued to MR by the control sequencer. Two functions work in concert in order to achieve this behavior: a *goal interpreter*, which uses the best estimate of system state and the system model to choose a highest reward goal state that satisfies the configuration goal, and a *reactive planner*, which uses the best estimate of system state and the goal state chosen by the goal interpreter to generate the next command necessary to achieve the specified goal state [3]. The model-based reasoning algorithms used by Titan are common to both the ME and MR processes.

3 Task Overview

3.1 Motivation

With the advent of model-based engineering and model-based design, the task of developing complex systems has become less error prone and more accurate. While the process of generating requirements in a model-based fashion certainly is an improvement over the traditional 'shall' statement form of requirements documentation, there is still a manual step involved in creating a software implementation from a set of requirements expressed as a set of models. It is natural, then, to ask if it is possible to develop our models in such a way that they are no longer simply descriptive, but also executable; that is, can the models be captured such that they can be provided as input to a computer program that will use them to perform some interesting system function?

This task came about in an attempt to show the potential benefits associated with applying an integrated MDS/Titan solution to the problem of designing and developing control software for high-performance resource-constrained spacecraft. As discussed in Section II, the MDS architecture is model-centric; historically, this has implied a manual translation of models into code. By introducing Titan, a model-based executive which embodies the model-based programming paradigm, into the MDS architecture, the 'holy grail' of software systems engineering becomes one step closer: generating one set of models which serve as both requirements for, and inputs to, the executing system.

There are other benefits to be considered as well. By focusing on the contents of the model rather than implementation issues associated with translating models to code, the bulk of the development effort can now be focused on getting the domain models right, rather than on software issues. In addition, any assumptions about the domain are made explicit in the models, not embedded in the code. This in itself is an important benefit, as it makes the task of reviewing the domain models for accuracy much more effective. Situations where "the code is the model", which can be quite dangerous, are avoided. With a model-based programming driven process, the "model is the code".

Lastly, this task can potentially lead to further advances in the application of model-based programming to embedded flight software development. Once model-based programming has been introduced and validated within a flexible state-based architecture, the potential for infusion of other model-based software technologies is increased; for example, new reasoning algorithms can be tested, or model-based programming solutions may be applied to other domains, such as activity planning and execution.

3.2 Objectives

The objectives for the task fall into three major areas of concern. First, as stated in the motivation, this task aims to perform an integration of both the ME and MR capabilities of Titan's deductive controller into a small but realistic MDS adaptation. Second, this task seeks to provide an assessment of the applicability of the integrated capability to future autonomous exploration missions of all sizes and classes based on the outcome of this activity. Finally, a set of guidelines and recommendations for a successful approach to verification and validation of model-based programs will be produced. This paper describes an initial step toward the achievement of these objectives.

3.3 Approach

The activities performed during the task were guided by the approach. In summary, five main work areas were defined:

1. Compatibility analysis of MDS and Titan;
2. Extension of the MDS frameworks to support integration of the Titan engine;
3. State Analysis of the example system and development of the supporting models;
4. Development of the MDS software adaptation; and
5. Testing of the integrated system.

Each of these tasks is discussed in detail below.

3.3.1 Compatibility Analysis

In order to determine the feasibility of the integration task, the first step was to perform a comparison of the MDS and Titan architectures and determine the level of compatibility between the two. An unclear or intractable mapping between the two domains would represent a formidable challenge to the integration of these architectures. After a preliminary analysis, however, a clear mapping between the two domains emerged where the core concepts in Titan correspond to specializations of core concepts in MDS. Briefly, the relationship can be characterized as follows:

- While State Analysis accomodates any appropriate model representation, Titan requires a particular representation, factored POMDPs, as presented in [3]. However, Titan's state-based model representation is wholly compatible with the requirements placed on models by State Analysis.
- An assignment to a Titan observation variable corresponds to a measurement in MDS, and an assignment to a Titan control variable corresponds to a command in MDS.
- Titan's notion of system state, which consists of an assignment to each state variable in the system, corresponds to a set of MDS state variables. While Titan represents state estimates as discrete value assignments to state variables, MDS represents state estimates in terms of *state functions*, which are continuous representations of state over time. Titan's discrete representation of state estimates corresponds to a constant state value persisting from one execution cycle to the next, a pattern which maps nicely to a constant state function in MDS.

- The functionality provided by ME replaces the hand-coded estimation algorithms of a typical MDS adaptation; similarly, the functionality provided by MR replaces the hand-coded control algorithms of past adaptations.

- Titan uses *configuration goals* to express intent. Configuration goals are constraints on the desired system state, and inherently represent both transition- and maintenance-type behaviors. MDS's *goals* are a more general representation, and may also be used to capture either transition behavior (e.g. turn rover wheel to $+90°$) or maintenance behavior (e.g. hold rover wheel at $+90°$). Although the MDS representation is more general, the Titan representation of intent is still consistent with that of MDS.

3.3.2 Framework Development

For this initial integration effort, there was not a significant amount of framework development performed; rather, most of the interface with Titan was relegated to the application-specific adaptation code. The first step in the framework development was assessment of the existing APIs to ensure sufficiency for integration of the Titan engine. The MDS framework APIs were found to be appropriate, without requiring extension or modification. For the purposes of this task, the Titan engine was built as an external library and owned by adapted classes. Future plans are to keep the Titan engine as an external library, to reduce framework dependencies on externally developed software, but to create a specialized framework achiever class which will own the instance of the engine and provide skeletons of the estimation and control methods required by the MDS architecture.

The second major portion of the framework development effort was developing the MDS interface to the Titan engine. From a design perspective, this activity was quite straightforward. Since Titan provides functionality nearly identical to MDS estimators and controllers, it was ideal to simply 'hide' the Titan interface inside MDS estimator and controller components; this is illustrated in Figure 4. These components in turn contains code to negotiate the interface with the Titan engine, in place of hand-coded achiever algorithms.

3.3.3 State Analysis and Modeling

The demonstration system chosen for the task was an Inertial Measurement Unit (IMU) and associated power switch. The task was able to leverage work performed on an existing MDS prototype performed in support of a Mars lander mission. With Titan integrated into the architecture, some modifications to the State Analysis artifacts were necessary:

- The existing State Effects Diagram had to be modified to break out some state variables that were previously combined. Specifically, the IMU operational mode and health were separated into a distinct state variables, as were the power switch operational mode and health state variables (see Figure 2).

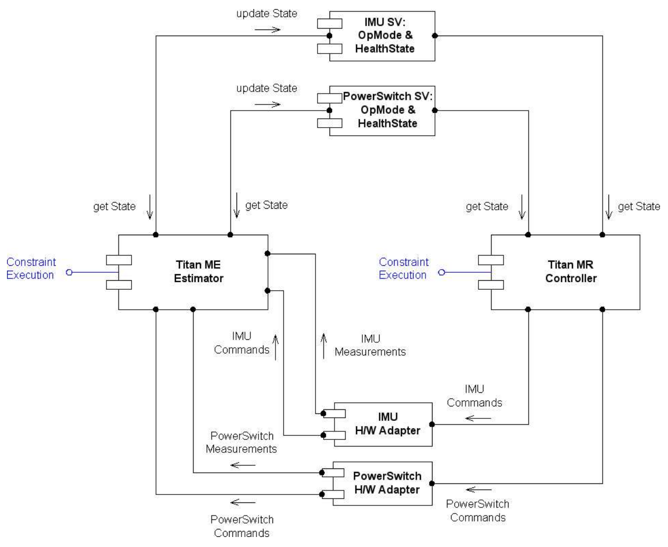
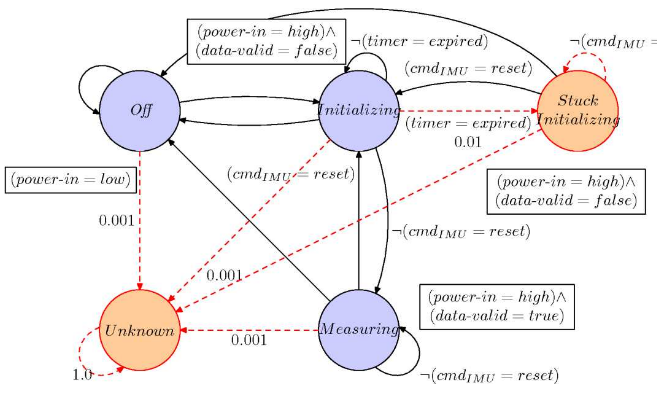


Figure 4. MDS/Titan Collaboration Diagram showing the main software components and the interfaces between them.

- The software collaboration diagram required modification due to the consolidation of estimation and control functionality for all SVs into one controller and one estimator component (see Figure 4).

- The previously developed estimation and control algorithms were no longer necessary, as Titan now performs the functionality that was previously contained in the hand-coded estimators and controllers.

The modeling portion of the task focused on the development of the subsystem models that Titan uses for the purposes of mode estimation and reconfiguration. Four models were developed for this system: a power switch operational model, and a power switch health model, an IMU operational model, and an IMU health model. The last two models are depicted in Figures 5 and 6. These models, expressed as factored POMDPs, represent state values as circles; it is important to note that both *nominal and off-nominal* (i.e., fault) *state values* are captured here. Each state value may define a *modal constraint* that must hold in order to transition to and remain in that state; modal constraints are contained in text boxes next to the associated state value. Each state may also have an associated reward. Solid arrows represent *transitions* between nominal state values, and are conditioned on the state guard expressions. Dashed arrows represent possible transitions to off-nominal state values. In addition to transition guard conditions, each transition has an associated probability. More detail on the formal semantics of the models is provided in [3].

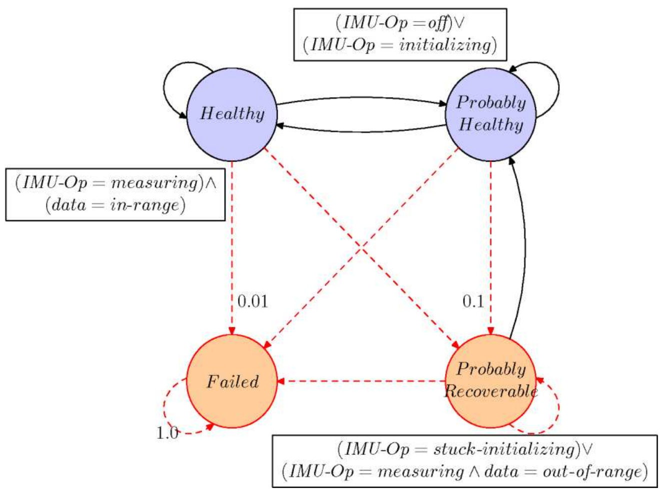


$(power\text{-}in = high) \wedge$
$(data\text{-}valid = false)$
$\neg(timer = expired)$
$\neg(cmd_{IMU} =$
$(cmd_{IMU} = reset)$
Off
Initializing
Stuck Initializing
$(timer = expired)$
0.01
$(power\text{-}in = low)$
$(cmd_{IMU} = reset)$
$(power\text{-}in = high) \wedge$
$(data\text{-}valid = false)$
0.001
$\neg(cmd_{IMU} = reset)$
0.001
$(power\text{-}in = high) \wedge$
$(data\text{-}valid = true)$
Unknown
Measuring
0.001
1.0
$\neg(cmd_{IMU} = reset)$

Figure 5. IMU Operational Mode Model [4]

3.3.4 Adaptation Development

The adaptation development was focused on developing estimator and controller components that use the Titan engine, rather than relying on hand-coded estimators and controllers. In addition, since Titan computes estimates for all state variables in a single computational cycle, several estimator and controller components were condensed into one estimator and one controller, as specified by the State Analysis (recall Figure 4). ME and MR use common model-based reasoning functions, so the interfaces were designed such that one instance of the Titan engine is shared between both achiever components. The estimator provides Titan with the latest state estimates as well as measurement and command data, then requests a state update. Using the models plus the state, measurement, and command information, Titan returns a set of state vectors, ordered by likelihood, with assignments for all state variables. The estimator then uses the most likely state vector it receives from Titan to update the associated state variables. The controller performs similar steps: first, a configuration goal and the latest state estimates are provided to Titan. Next, the controller asks Titan whether any commands need to be sent in order to drive the system to a least-cost goal state. The controller will then take any action needed based on the result of the request (i.e., issue the specified command). These estimator and controller interface algorithms are concisely summarized in Figure 7. The benefits of this scheme are clear, even during the design and implementation phase. Since Titan is responsible for estimating and controlling several state variables, the number of software components to be developed is reduced. In addition, since estimation and control logic is effectively provided by Titan, no detailed algorithms for estimation and control need to be written, tested, and debugged. In fact, the logic required to interface with the Titan engine is quite generic and application-neutral, allowing developers to simply instantiate an implementation of a general 'Titan Achiever' component in place of a handcrafted estimator or controller with little specialization (see Figure 7). This amounts to a significant savings in terms of development time.



$(IMU\text{-}Op = off) \vee$
$(IMU\text{-}Op = initializing)$
Healthy
Probably Healthy
$(IMU\text{-}Op = measuring) \wedge$
$(data = in\text{-}range)$
0.01
0.1
Failed
Probably Recoverable
1.0
$(IMU\text{-}Op = stuck\text{-}initializing) \vee$
$(IMU\text{-}Op = measuring \wedge data = out\text{-}of\text{-}range)$

Figure 6. IMU Health Model [4]

3.3.5 Testing

The test portion of the effort was based around two main thrusts: software verification and model verification. The tests were designed to ensure that from a software standpoint, all of the interfaces between Titan, the MDS frameworks, and the adaptation code were properly defined and implemented. All testing was performed against a software simulation, which allowed for testing of both nominal and fault behaviors. Success criteria are based on the models and observed behavior of the system without Titan in the loop – that is, using hand-coded achievers. From the standpoint of model verification, the tests were designed to root out any latent issues with the models themselves. During the course of this effort, the most common modeling error observed was unrealistically high probabilities associated with several of the off-nominal states, an error which sometimes resulted in a state variable being improperly diagnosed to be in an off-nominal state. These errors were discovered during test and the probabilities were subsequently tuned to values which more closely represented likelihoods associated with the particular failure modes.

Each test case is based on a single operational scenario, and therefore uses the same activity plan, expressed in terms of a *goal network* (refer to [1] for more information on the details of activity plans and goal elaborations). The plan places a goal on the IMU operational mode to transition to and maintain the "Measuring" state; this goal elaborates into subgoals on the IMU health state variable, and the operational mode and health state variables of the power switch, constraining them to transition to and maintain "Healthy", "Closed", and "Healthy" states, repectively. The plan is constrained to stay active for a duration of 120 seconds. Using this activity plan, five test cases were defined for this task, each designed to exercise a particular behavior of the system (including failure modes).

- **No Fault:** This is the most basic system test, designed to exercise nominal operation of the system. The nom-

```
subroutine Estimate()
{
    Provide observation data to Titan
    Provide command data to Titan
    Request state update from Titan
    Update MDS State Variables with new
      state estimates
}

subroutineControl()
{
    Map MDS Constraint to Titan
      Configuration Goal
    Query all relevant MDS State Variables
    Provide latest state estimates to Titan
    Request new command from Titan
    if command to be issued
        translate Titan command to hardware
          command
        send command to hardware
    end if
}
```

Figure 7. Pseudocode for Titan Estimator and Controller Interfaces

inal plan, described above, is put in place and allowed to execute to completion. Nominal measurements are provided by the simulation throughout the test.

- **Fault Case 1:** Test designed to verify proper system behavior in response to a recoverable fault case where the IMU fails to exit its initalization state. The IMU is induced to produce initialization data, via a fault injection through the simulation, even after the alloted initialization period has expired. Recovery is accomplished by issuing a reset command to the IMU.

- **Fault Case 2:** Test designed to verify proper system behavior in response to a recoverable fault case where the IMU produces out-of-range data during nominal operation. The IMU is induced to produce out-of-range data, via a fault injection through the simulation, sometime after the IMU has entered the measuring state. Recovery is accomplished by issuing a reset command to the IMU.

- **Fault Case 3:** Test designed to verify proper system behavior in response to a non-recoverable fault case where the IMU never exits its initalization state. The IMU is induced to produce initialization data, via a fault injection through the simulation, even after the alloted initialization period has expired. The fault injection is left applied such that attempts to repair the IMU will have no effect.

- **Fault Case 4:** Test designed to verify proper system behavior in response to a recoverably tripped IMU power switch fault. The IMU power switch is tripped open 30 seconds after test start via an overcurrent fault injection through the simulation, resulting in a loss of power to the IMU. Recovery is accomplished by issuing a command to open the switch (to "clear" the tripped condition), followed by a command to close the switch (to power on the IMU).

The above set of test cases exercises both nominal and off-nominal operational modes. For a follow-on effort, it will be desirable to define a larger set of test cases in order to test more exotic failure modes and provide a broader set of data with which to characterize the behavior of a more complex system, containing more state variables.

4 Results

The implementation was quite straightforward. Most of the issues associated with integrating Titan into an MDS adaptation were revealed during integration of ME; as a consequence, the MR integration progressed very quickly.

Several interesting challenges were uncovered during the task. Some of these issues were resolved in the implemented system; some were resolved in principle but not implemented in the demonstration; the rest were noted as issues to be addressed by future work. A number of these issues are discussed below.

- **Issue 1:** How to accomodate the "centralization of functionality" provided by Titan within the MDS architecture?

- **Resolution:** The centralization of functionality that naturally accompanies this approach is not a problem as far as MDS is concerned, as long as we do not violate any of the stated architectural principles, notably:

  - All state information is stored in state variables – achievers may not store any local state.
  - State variables have only one associated estimator and controller. Only the associated estimator may update that state variable.
  - Estimators and controllers may be responsible for multiple state variables, as long as they are the only estimator or controller associated with that state variable.

  Our implementation has adhered to these fundamental principles.

- **Issue 2:** Titan adopts the notion of an *approximate belief state*, which is an assignment of values to all state variables and an associated likelihood. How does this notion fit into the MDS architecture, where state variables are the single source of state value and uncertainty information?

- **Resolution:** One possible resolution would be to collapse all state variables into a single "system state variable" which could capture all *K* leading estimates. Uncertainty would be represented by the approximate belief state probabilities associated with each of the *K* estimates. However, a more modular solution that retains the individual state variables within the architecture may be desirable. Since we have access to the *K* most likely system state estimates, a particular state variable's estimate can be computed by finding all system state estimates which contain that same state assignment and adding up the associated probabilities (this provides an upper bound on the true probability of being in that state). This information can be used to construct the associated uncertainty for that state assignment. The current implementation does not perform this computation; instead, only the assignments corresponding to the most likely system state estimate are captured in the state variables, and assumed to be perfectly known or unknown. The limitation of this approach is that if, when presented with some amount of observation data, Titan determines that the current most likely estimate is in fact wrong and a new state estimate becomes most likely, this sudden change of course appears as a sharp discontinuity to the rest of the control system. Future work would implement the more accurate method of tracking probabilistic uncertainty.

- **Issue 3:** Titan operates only on discrete states. How do we integrate Titan with portions of the system that deal with continuous state variables?

- **Resolution:** Another benefit of this task is providing a solution to this issue. MDS allows for integration of various estimation and control approaches for different state variables such as Kalman filters, hypothesis testing, PID control, etc., as dictated by the needs of the application. One reasonable solution is to simply allow Titan to handle a subset of the system's discrete states and create traditional achievers for the continuous states. Another possible solution is provided by active research in the area of *hybrid model-based programming*; for example, [9] describes an any-time algorithm for generating estimates for discrete modes *and* continuous state variables. Integrating this type of hybrid state estimation capability is identified as a possible direction for future work.

- **Issue 4:** During testing of the integrated ME and MR functionality, several of the tests began to fail due to a faulty estimation of the power switch health state variable as being in an UNKNOWN state, despite the fact that consistent power switch observations were being provided to Titan. The failure was observed to occur less than a second after test start, before any activities had commenced.

- **Resolution:** This issue illustrated a potential limitation of the current Titan design; namely, Titan does not properly handle some types of dependencies between state variables. This limitation resulted in an inability to discriminate between nominal power switch behavior and the UNKNOWN state (a low probability state that is consistent with all behavior). In the absence of a state dependency in the model, the BFBSU algorithm would be able to properly discriminate between these states. This issue was exacerbated by the presence of some modeling irregularities in the IMU and power switch models developed for this task. The issue was mitigated as follows. The probability associated with the UNKNOWN state was reduced to a more realistic value of $10e^{-6}$; this effectively delayed the time that it would take for the estimate containing the UNKNOWN state to become the most likely estimate, long enough such that a transition would occur, causing the estimates to change and the probability of the failed estimate to drop as a result. To more thoroughly address this issue, future work will need to focus on two areas:

  - characterization of exactly which types of dependencies are problematic for the existing algorithm, and modify the Titan algorithm to accomodate such interdependencies between state variables.

  - development of a set of modeling guidelines to ensure that the source models properly express the physical dependencies of the system under control.

- **Issue 5:** MDS allows for the specification of knowledge goals, i.e., constraints on the level of uncertainty in a state estimate. How are knowledge goals interpreted by Titan, which currently has no corresponding notion?

- **Resolution:** Titan represents estimate uncertainty in terms of a belief state probability on a given state variable. Therefore, a knowledge goal could be expressed in terms of a minimum percent likelihood of a particular state value. As suggested above, the computation of the probability of each state variable assignment can be performed by adding the probabilities associated with every state estimate in the belief state that contains this particular assignment. Goal satisfaction can then be verified by comparing this computed probability with the level of uncertainty specified in the goal. One might even envision the elaboration of active control subgoals in support of a transition goal requesting a higher quality estimate, resulting in "active probing" of the system to increase the certainty in the state estimate. Another solution would be to increase the number of tracked state estimates *K* in order to improve the quality of the likelihood estimate.

5 Conclusion

MDS and Model-based Programming are proving to be a natural fit. The state-based systems engineering methodology embodied by the State Analysis process considers models to be central to the design of high-assurance, fault-tolerant control system software. However, the translation of models developed during State Analysis into executable code is still a manual process, which can be arduous and error-prone. Augmenting the MDS frameworks with a state-of-the-art, model-based reasoning engine can provide both systems and software engineers with the capabilities necessary to create reliable control software, and do it more rapidly. While this particular task was limited in scope, the results are quite promising. Future work will expand the scope of the demonstration and further characterize the applicability, benefits, and limitations of the integrated capability for space exploration missions.

Acknowledgments

This research was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration and funded through the Director's Research and Development Fund program. The authors would like to thank the following people for their assistance with this task: Dave Wagner for his help with issues related to the MDS frameworks during the initial integration; Len Charest, who performed the initial integration of the mode estimation capability; and Paul Elliott for his help integrating the Titan software.

References

[1] M. D. Ingham, R. D. Rasmussen, M. B. Bennett, and A. C. Moncada, "Generating requirements for complex embedded systems using State Analysis," *Proceedings of the 55th International Astronautical Congress*, October 2004, paper #IAC-04-IAF-U.3.A.05.

[2] D. Dvorak, R. Rasmussen, G. Reeves, and A. Sacks, "Software architecture themes in JPL's Mission Data System," *Proceedings of the IEEE Aerospace Conference*, 2000.

[3] B. C. Williams, M. D. Ingham, S. H. Chung, and P. H. Elliott, "Model-based programming of intelligent embedded systems and robotic space explorers," *Proceedings of the IEEE: Special Issue on Modeling and Design of Embedded Software*, 2003.

[4] O. B. Martin, "Accurate belief state update for probabilistic constraint automata," Master's thesis, Massachusetts Institute of Technology, 2005.

[5] D. Bernard *et al.*, "Spacecraft autonomy flight experience: The DS1 Remote Agent Experiment," *Proceedings of the AIAA Space Technology Conference and Exposition*, 1999, paper #AIAA-99-4512.

[6] M. S. Feather, L. M. Fesq, M. D. Ingham, S. L. Klein, and S. D. Nelson, "Planning for V&V of the Mars Science Laboratory rover software," *Proceedings of the IEEE Aerospace Conference*, 2004.

[7] O. B. Martin, B. C. Williams, and M. D. Ingham, "Diagnosis as approximate belief state enumeration for probabilistic concurrent constraint automata," *Proceedings of the Twentieth National Conference on Artificial Intelligence*, 2005.

[8] O. B. Martin, S. H. Chung, and B. C. Williams, "A tractable approach to probabilistically accurate mode estimation," *Proceedings of the Eigth International Symposium on Artifical Intelligence, Robotics, and Automation in Space*, 2005.

[9] M. W. Hofbaur and B. C. Williams, "Mode estimation of probabilistic hybrid systems," in *Hybrid Systems: Computation and Control*, C. Tomlin and M. Greenstreet, Eds., vol. 2289. Springer-Verlag, 2002, pp. 253–266.

[10] M. D. Ingham, "Timed Model-based Programming: Executable specifications for robust mission-critical sequences," Ph.D. dissertation, Massachusetts Institute of Technology, 2003.