MODEL-BASED AUTONOMY FOR THE NEXT GENERATION OF ROBOTIC SPACECRAFT

Fesq, L.(1); Ingham, M.(1); Pekala, M.(2); Van Eepoel, J.(1); Watson, D.(2); Williams, B.(1)
(1) Space Systems Laboratory, Massachusetts Institute of Technology
(2) Applied Physics Laboratory, Johns Hopkins University
Contact: {fesq, ingham, vanny, williams}@mit.edu; {mike.pekala, dave.watson}@jhuapl.edu

ABSTRACT

A novel approach to the design of reactive embedded software systems, called *model-based autonomy*, is generating significant interest in the space systems community. The goal of the model-based approach is to automate onboard sequence execution by tightly integrating goal-driven commanding with fault detection, diagnosis, and recovery capabilities. This paper describes *Titan*, a system-level autonomy framework that expands upon previous model-based approaches to configuration management, such as the Livingstone mode identification and reconfiguration system, which was demonstrated on the Deep Space One spacecraft. Titan is a model-based executive that estimates current spacecraft modes, detects and repairs failures, and executes commands, all within a fast sense-decide-act loop. This paper discusses the model-based software technologies implemented within the Titan executive. It describes a case study of the deployment of Titan to a representative space mission, including an overview of the various component models assembled, as well as the scenarios generated to verify proper execution of the software. Finally, it provides results from initial tests of the Titan implementation on these scenarios and models.

INTRODUCTION

NASA’s vision of an end-to-end autonomously operated space flight system is inspiring the development of enabling technologies for highly robust spacecraft. Over the past few years, a new approach to the design of reactive embedded software systems called *model-based autonomy* has generated significant interest in the space systems community. The goal of the model-based approach is to automate onboard sequence execution by tightly integrating goal-driven commanding with fault detection, diagnosis, and recovery capabilities. Model-based autonomy has been deployed in various aerospace applications, including the Deep Space One (DS-1) mission,[1] and ground testbeds for the Space Interferometry Mission,[2] the X-34 and X-37 rocket planes, and an in-situ propellant production system.[3]

This paper describes a system-level autonomy framework that significantly expands upon previous model-based approaches to fault protection, such as the Remote Agent mode identification and reconfiguration system (Livingstone[4]), which was flown onboard the DS-1 spacecraft.[1] This autonomy framework, named *Titan*, is a model-based executive that is capable of estimating current spacecraft modes, detecting and repairing failures, and executing commands, all within a fast sense-decide-act loop. It leverages techniques from several fields of artificial intelligence research, including model-based reasoning, Markov modeling and constraint programming.

The Titan model-based executive adopts the notion of *model-based programming*[5] as an approach to writing software for embedded reactive systems. The underlying principle is that embedded control software can be written by asserting and checking states which may be “hidden”, i.e. not directly controllable or observable, rather than by operating on observable and control variables.

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 probabilistic models of the plant's nominal behavior and common failure modes.

A model-based program is executed by automatically generating a control sequence that transitions the physical plant to the states specified by the control program. These specified states are called *configuration goals*. Program execution is performed using the Titan executive, which repeatedly generates the next configuration goal, and then generates a sequence of control actions that achieve this goal, based on knowledge of the current plant state and plant model.

The goal of this paper is to provide an overview of the Titan executive, and to illustrate its application to a representative space mission. In particular, this paper presents results from a demonstration of Titan performed in the context of a NASA New Millennium Space Technology 7 Autonomy (ST7-A) concept definition study.[6] A successful demonstration consists of showing that Titan can provide the capabilities identified as critical for robust execution of a real space mission.

This paper first provides a description of the model-based technologies implemented within the Titan executive. It describes a case study of the deployment of Titan on the ST7-A space mission, including an overview of the various plant models and control programs assembled, as well as the scenarios generated to verify proper execution of the engines. Finally, it discusses initial tests of the Titan implementation on these scenarios and models, highlighting the demonstration of critical capabilities for robust execution.

MODEL-BASED SOFTWARE OVERVIEW

Titan Overview

Titan is a model-based executive that estimates current spacecraft modes, detects and repairs failures, and executes commands within a fast sense-decide-act loop. It is intended to interact with a system-level planning capability (which may be ground-based, or included as part of the onboard software architecture), by executing scheduled activities in the mission plan. It commands low-level hardware and spacecraft subsystems.

As shown in Figure 1, Titan is composed of two modules, 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 be placed in. The deductive controller is responsible for estimating the plant's most likely current state based on observations from the plant (*mode estimation*), and for issuing commands to move the plant through a sequence of states that achieve the configuration goals (*mode reconfiguration*).

Titan provides several advantages over existing onboard execution systems. Similar to the Remote Agent Executive,[7] it uses a combination of scripted and deduced actions, but Titan's model-based control sequencer design allows for a simpler interaction between script execution and the deductive control engine. This design lends itself to goal-oriented commanding, and control code that is highly modular, reusable, and more easily verified by spacecraft engineers. The tight coupling of state determination and reconfiguration within the deductive controller allows for on-the-fly recovery actions to be inferred and executed in real time, for seamless continuation of operations in the event of failure or anomalies. Finally, the high degree of plug-and-play modularity within the Titan system, at both the model and module level, will allow for high portability between missions.

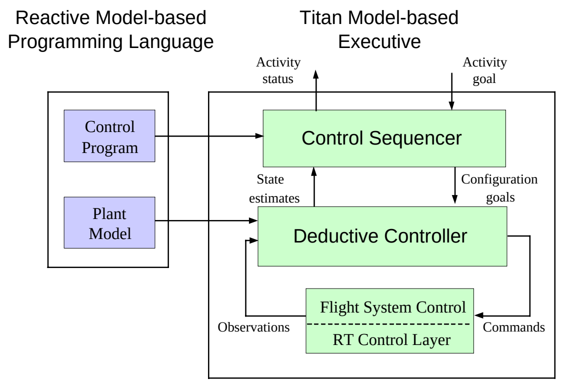

Figure 1. Titan Architecture

Control Sequencer Overview

The control sequencer takes in activity goals from the ground or an onboard system-level planner, such as ASPEN,[8] Europa[9] or Kirk.[10] Its role is to decompose these system-level activity goals into lower-level configuration goals for the hardware components and subsystems. Each activity goal invokes a control program written in the Reactive Model-based Programming Language (RMPL). The control program can be viewed as a deterministic, executable specification of the desired spacecraft behavior, expressed in terms of the spacecraft's state variables. As the sequencer executes a control program, it reads the current spacecraft state from, and issues appropriate configuration goal states to, the deductive controller. Technical details on RMPL and the control sequencer are presented in reference 5.

Deductive Controller Overview

The deductive controller (Figure 2) takes as inputs the plant model, a sequence of configuration goals, and a sequence of observations. It uses the same plant model to deduce the system state from the observations, and to figure out how to achieve the configuration goal states issued by the control sequencer. It generates a sequence of most likely plant state estimates and a sequence of control actions (commands). The sequence of state estimates is generated by a process called *mode estimation* (ME). The process of determining which control actions should be issued in order to achieve a given configuration goal is called *mode reconfiguration* (MR). The deductive controller has been described extensively in references 4 and 11; only a high-level description of the ME and MR engines is provided here.

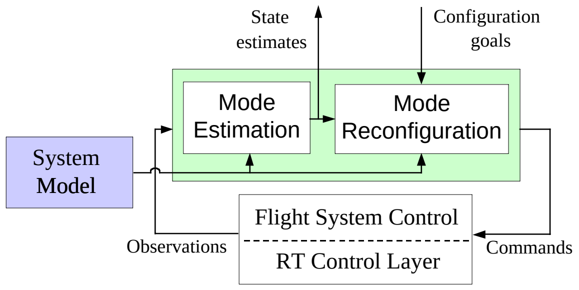

Figure 2. Titan's Deductive Controller

Mode Estimation

The ME engine provides confirmation of commands, spacecraft state tracking, fault detection, and fault diagnosis. ME interprets real-time sensor telemetry by using qualitative observations in conjunction with an onboard state model of the system to estimate the current spacecraft state. It detects faults as discrepancies between observed and expected system behavior, based on the model and commands that have been issued. When a fault is detected, ME reasons through the probabilistic plant model to determine the most likely current state of the system that is consistent with the new observations.

More specifically, ME incrementally tracks the set of system state trajectories that are consistent with the plant model, the sequence of observations and the control actions. The ME process is framed as an instance of Hidden Markov Model belief state update, which computes the probability associated with being in a certain state at each time step. Since the size of the set of possible current states is exponential in the number of components, computational resource limitations only allow a small fraction of the state space to be explored in real time. ME tracks only the most likely states using the OpSat optimal constraint satisfaction engine.[12]

Mode Reconfiguration

The MR engine achieves configuration goals by issuing a sequence of appropriate hardware commands, based on the current estimated state from ME, and the onboard state models of the hardware components. MR accomplishes this through two capabilities, the *goal interpreter* (GI) and *reactive planner* (RP). GI uses the plant model and the most likely current state to determine a reachable target state that achieves the configuration goal, while minimizing cost. RP takes the target state and a current mode estimate, and generates a command sequence that moves the plant to this target. RP generates and executes this sequence one command at a time, using ME to confirm the effects of each command. It should be noted that GI also uses OpSat in its search for a minimum-cost target state. Because RP can generate repair plans for faults requiring multiple-step repair actions, the MR engine provides increased repairable fault coverage over previous model-based recovery systems. RP, also called *Burton*,[11] is a sound, complete planner that generates a control action of a valid plan in average case constant time.

MISSION APPLICATION

Scenario Overview

The ST7-A study used a reference mission concept consisting of a fully autonomous science spacecraft in low earth orbit. The spacecraft incorporated multiple science instruments and standard bus functionality, including data recording, attitude control, and ground communications.

To demonstrate the applicability of model-based autonomy to real-world missions, plant models and control programs for a number of scenarios depicting ST7-A operations were developed. Two of these scenarios will be described in this paper.

Scenario 1

Scenario 1 captures the operation of transmitting stored data to the ground. This scenario, called DownlinkDataBlock, demonstrates the capabilities of the control sequencer and its interaction with the deductive controller. The control sequencer is sent an activity goal to downlink stored science data to the ground. The sequencer responds by initiating a ground communication activity in order to play back the stored data and thereby free up onboard data storage space. A ground communication activity involves:

1. determining which onboard omni antenna will have line-of-sight coverage to the ground station;
2. configuring the onboard communications system to transmit real-time telemetry out the appropriate omni-directional antenna;
3. receiving communication confirmation from the ground;
4. transmitting stored data from the solid-state data recorder to the ground;

5. idling the transmitter once data transmission is complete, and reporting success of the DownlinkDataBlock activity.

This example uses a simplified model of the communication subsystem for the ST7-A spacecraft as shown in Figure 3. The subsystem consists of two omni-directional antennas that are diametrically opposed, providing 4-pi steradian coverage. A single transmitter can be connected to either of the two omni antennas via a switch. The position of the switch is sensed and fed back to ME to determine configuration status. When ME diagnoses off-nominal system behavior, such as the switch being stuck in the wrong position, the sequencer is notified, which in turn fails the DownlinkDataBlock activity and notifies a system-level planner such as Kirk. The planner would then replan and invoke an alternate control program, possibly involving spacecraft reorientation, in order to achieve the downlink goal.

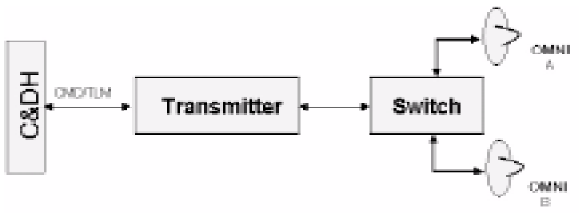

Figure 3. Scenario 1 Block Diagram

Scenario 2

Scenario 2 demonstrates the capabilities of the ME and MR engines by providing more sophisticated models for components of a command and data handling subsystem and simulating nominal and failed operations of these components. The components in Scenario 2 consist of a prime and redundant 1553 Bus Controller (BC), a single 1553 command/data bus, two remote terminals (RTs), and two devices each connected to one of the remote terminals, as shown in Figure 4. The bus controllers are mutually exclusive: only one BC is on and controlling the bus at any given time.

This scenario begins with BC-A controlling the bus, and with the RTs and devices turned off. The Titan sequencer receives an activity goal to activate the devices, which involves systematically powering on the RTs and the devices. As MR sends commands to power on the boxes, ME monitors the system behavior to determine nominal vs. off-nominal conditions. Upon sensing off-nominal behavior, ME notifies MR, which in turn attempts to achieve the configuration goal through hardware reconfiguration; e.g., resetting or power-cycling a non-responsive bus controller. By evaluating observables in the system such as comm/no-comm status from the devices, ME diagnoses the state of the individual components, distinguishing between BC failures and RT/device failures. If the primary BC is determined to have failed in a non-recoverable way, control of the bus passes to the backup BC.

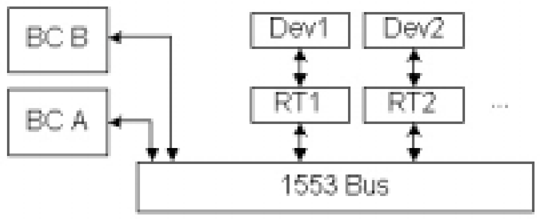

Figure 4. Scenario 2 Block Diagram

Plant Model Overview

A spacecraft engineer implements plant models for the Titan deductive controller using Livingstone's modeling language, called the Model-based Programming Language (MPL). MPL provides constructs for defining plant models, which are compiled into a format suitable for processing by the Titan executive at runtime.

Plant modeling begins with the development of individual component models using a first-principles approach. These component models are then logically combined to form the complete system model. The Titan executive uses this aggregate model at runtime to perform system-wide inference. This component-based approach to model development frees the spacecraft engineer from having to explicitly anticipate and encode all possible system-wide interactions a priori.

Figures 5 and 6 depict graphical representations of Titan plant models. These individual component models are defined in terms of operational states or *modes*, which may be characterized as "nominal" or "faulty". In the figures, modes are represented by labeled ellipses. A component occupies precisely one mode at any single point in time. Each mode is defined in terms of its *modal constraints*, which are propositional logic statements specifying the consistent plant behavior for that mode. Plant behavior is gauged primarily in terms of *observations* that have been gathered from system sensors. *Transitions*, which are the arcs in the component model diagrams, define the allowable trajectories between component modes. *Nominal, or commanded, transitions* are explicitly encoded by the spacecraft engineer and have associated with them a *guard* that specifies when these transitions are enabled. *Fault transitions* are implicit in the model, and use the probabilities and modal constraints associated with fault modes, rather than guards, to determine possible resultant states. The included figures follow the convention of using a solid line to depict nominal transitions and a dashed line to depict fault transitions. Labels on the arcs correspond to the guard associated with the transition.

Good modeling practice dictates that component models should always contain an “unmodeled” or “unknown” fault mode that has an extremely low probability and no modal constraints. This is used as a fallback in the event of a completely unanticipated fault not covered by an explicitly encoded fault mode. For brevity, this mode is sometimes omitted from the graphical representation of the component model.

Our plant models for Scenario 1 consist of a Transmitter component and a Switch component. The Transmitter component represents the aggregate behavior of communication subsystem electronics (including transponder, diplexer, and solid-state recorder hardware, for example). The Transmitter has three nominal modes of operation: “idle”, in which the communication link is not established, “streaming-rt-telem”, in which a low-rate communication link with the ground is established, and “downlinking-data”, in which playback data is being transmitted to the ground at a high data rate. The Switch can be in one of two nominal modes, “enable-OmniA” or “enable-OmniB”, which correspond to the switch positions enabling OmniA or OmniB, respectively, for communication with the ground.

The Switch model, depicted in Figure 5, is also assumed to have two possible failure modes, *stuck-at-A* and *stuck-at-B*, in which the switch is stuck in position and cannot be reset.

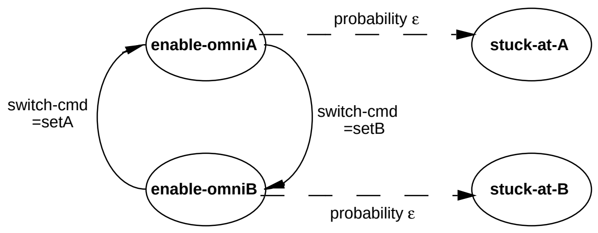

Figure 5. State transition model for Switch

The Transmitter and Switch component models developed for Scenario 1 are, by design, simple examples of first-principles model development. The internal logic is straightforward, and the models themselves map clearly to physical elements of the underlying plant. However, Titan’s plant modeling language is flexible and allows a spacecraft engineer to produce more complicated models. For example, the bus controller model developed for Scenario 2 is implemented using three interrelated component models: one “concrete” model of the bus controller itself and two associated “pseudo-component” models, which are used to record attempted recovery actions and do not have a physical counterpart in the underlying plant.

Figure 6 shows the concrete bus controller plant model, which has the nominal modes “on” and “off”, and has failures modes “resettable”, “power-cycleable”, “broken” and “unknown”. A bus controller in “on” mode is powered and is actively managing data traffic on the 1553 bus, while “off” mode corresponds to the powered down state. The more interesting aspect of this component is the way in which failures are modeled. In this case, “resettable”, “power-cycleable” and “broken” form an ordered sequence of cascading faults, ranked by likelihood. All three are observationally indistinguishable, and correspond to a lack of communications capability on the 1553 bus.

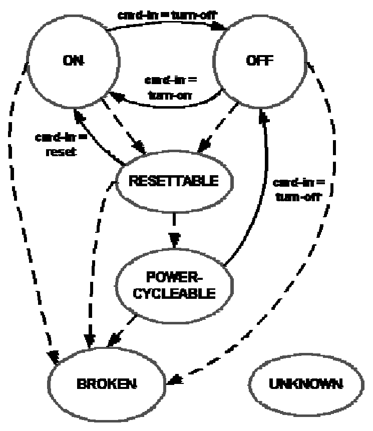

Figure 6. Bus Controller Component Model

In the event of a bus controller failure, the “resettable” mode will be selected as the most likely state by the deductive controller, as it has the highest probability. Once in “resettable” mode, the model defines a transition back to the “on” mode by means of issuing a “reset” command to the bus controller. However, issuing this “reset” command also has a side effect – one of the pseudo-component models associated with the bus controller will change state as well. Should this recovery action fail to restore communications, the pseudo-component state will indicate that a reset has already been attempted and a constraint on the “resettable” mode of the bus controller will preclude a repeated “resettable” diagnosis. Thus, estimation falls to the next most likely mode, which is "power-cycleable". Like "resettable", "power-cycleable" has an associated pseudo-component that tracks the number of repair actions that have been initiated from this state. If both the reset and power cycle repair actions fail to restore communications, the bus controller falls into the "broken" state, in which no further repair options exist.

The mechanism described above is one of numerous possible ways to encode this notion of cascading faults in Titan. This particular approach exhibits the property that it encodes aspects of the recovery policy directly within the component plant model. It also highlights the fact that component models are applicable to more abstract notions beyond the physical plant hardware. In addition to implementing other plant modeling approaches for handling cascading faults, the spacecraft engineer can elect to delegate this functionality to the control program instead. The Titan architecture and modeling languages provide the flexibility to select the most natural approach for a given application.

Control Program Overview

A control program is a deterministic, executable specification of the desired spacecraft behavior, expressed in terms of the spacecraft's state variables. It is written in a model-based programming language, such as RMPL. RMPL provides standard programming constructs for encoding various types of behavior: conditional branching, iteration, preemption, concurrency and sequential execution.

The RMPL control program in Figure 7 specifies state trajectories for the DownlinkDataBlock scenario. In addition to the state variables corresponding to the above-mentioned component modes, the control program also references other system state variables: 'omni-in-view' (captures the spacecraft's knowledge of which Omni antenna has the most direct view of the ground), 'ground-command' (captures the current command received from the ground), 'ssr-playback' (indicates completion of solid state recorder playback), and 'downlink-status' (signals the success or failure of the DownlinkDataBlock activity).

In order to execute the downlink activity, the following actions must be performed:

1. Set the Switch to enable the appropriate Omni antenna for downlink of streaming real-time telemetry (based on 'omni-in-view' info obtained from the attitude control subsystem, for example.)
2. If the Switch gets stuck in the wrong position, fail the DownlinkDataBlock activity.
3. Otherwise, set the Transmitter to start low-rate streaming of real-time telemetry so that the ground can establish the comm link.
4. When the "start-playback" command is received from the ground, start the high-rate downlink of onboard data.
5. When the data downlink is finished, set the transmitter to idle and report success of the DownlinkDataBlock activity.

The control program in Figure 7 highlights several different types of behavior:

- **conditional branching** – As indicated by the *if-thennext-elsenext* construct (lines 4 & 20), the control code must execute differently, depending on whether OmniA or OmniB is in view. Also, conditions on the switch-mode (line 7) and ground-command variables (line 13) are checked prior to initiating the streaming of telemetry and downlinking of data, respectively. This condition-checking is implicit in the *when-donext* construct.
- **iteration** – Should the above conditions on switch-mode and ground-command take some time to be achieved, the control program should wait for them to become true (lines 7 & 13), requiring an iteration on the condition check (again implicit in the *when-donext* construct). Also, the *always* construct is used in this program, to iteratively maintain the Transmitter in "streaming-rt-telem" and "downlinking-data" modes (lines 9 & 16).
- **preemption** – While the Transmitter is "downlinking-data", the control program needs to be watching for the state 'ssr-playback = complete' to become true, at which point the "downlinking-data" goal can be terminated (line 17). Such preemption is enabled by the *do-watching* construct.
- **concurrency** – As indicated by the *parallel* construct (line 5), the following tasks are performed concurrently: setting the Switch to the appropriate mode (line 6); checking for the correct Switch position prior to initiating telemetry streaming (line 7); checking for the Switch being stuck in the wrong position (line 11); and checking for receipt of the "start-playback" ground command (line 13) .
- **sequential execution** – As indicated by the *sequence* construct (line 14), the following tasks are performed sequentially: maintaining the Transmitter in "downlinking-data" mode until the 'ssr-playback = complete' notification is received (lines 15-17); setting the transmitter-mode to "idle" (line 18); and reporting success of the DownlinkDataBlock activity (line 19).

```
;; The following example uses these state variables and domains:
;;  - omni-in-view has values [omniA omniB]
;;  - transmitter-mode has values [idle streaming-rt-telem downlinking-data unknown]
;;  - switch-mode has values [enable-omniA enable-omniB stuck-at-A stuck-at-B unknown]
;;  - ground-command has values [no-cmd start-playback]
;;  - ssr-playback has values [incomplete complete]
;;  - downlink-status has values [undetermined succeeded failed]

DownlinkDataBlock()::
{do
   {sequence
     (downlink-status = undetermined)
     {if (omni-in-view = omniA) thennext
        {parallel
           (switch-mode = enable-omniA)
           {when (switch-mode = enable-omniA) OR (switch-mode = stuck-at-A) donext
              {do
                 {always (transmitter-mode = streaming-rt-telem)}
               watching (ground-command = start-playback)}}
           {when (switch-mode = stuck-at-B) donext
              (downlink-status = failed)}
           {when (ground-command = start-playback) donext
              {sequence
                 {do
                    {always (transmitter-mode = downlinking-data)}
                  watching (ssr-playback = complete)}
                 (transmitter-mode = idle)
                 (downlink-status = succeeded)}}}
      elsenext   ;; Similarly for the case where (omni-in-view = omniB)
      {…
…
 watching (downlink-status = failed) OR (downlink-status = succeeded)}
```

Figure 7. Control Program for DownlinkDataBlock Activity (ST7-A Scenario 1)

```
;; The following example uses these state variables and domains:
;;  - bc_a has values [on off resettable power-cycleable broken unknown]
;;  - bc_b has values [on off resettable power-cycleable broken unknown]
;;  - maintainBCstatus has values [sustaining failed]

MaintainOneBCOn()::
{do
   {sequence
     (maintainBCstatus = sustaining)
     {parallel
       {do
         {always
           {parallel
             (bc_a = on)
             (bc_b = off)}}
        watching (bc_a = broken) OR (bc_a = unknown)}
       {do
         {when (bc_a = broken) OR (bc_a = unknown) donext
           {always
             (bc_b = on)}}
        watching (bc_b = broken) OR (bc_b = unknown)}
       {when ((bc_a = broken) AND (bc_b = broken)) OR
             ((bc_a = broken) AND (bc_b = unknown)) OR
             ((bc_a = unknown) AND (bc_b = broken)) OR
             ((bc_a = unknown) AND (bc_b = unknown))
        donext (maintainBCstatus = failed)}}}
  watching (maintainBCstatus = failed)}
```

Figure 8. Control Program for maintainBCstatus Activity (ST7-A Scenario 2)

These types of behaviors are common to embedded programming. The key distinction in a model-based control program is the direct referencing of hidden state variables in the system. RMPL control programs may be viewed as specifications of deterministic state transition systems, which act on the plant by asserting and checking constraints expressed in propositional state logic. The propositions are assignments of state variables to values within their domains. Reactive constructs, such as *do-watching*, *when-donext* and *always*, allow flexibility in expression of complex system behavior and dynamic relations.

Figure 8 presents one of the control programs used in the context of Scenario 2. This program encodes the switch-over to 'bc_b' in the case of an unrecoverable or unknown failure of 'bc_a'. This example demonstrates the use of control programs to implement "state maintenance" activities intended to run continuously as background processes on the spacecraft processor. This is in contrast to the DownlinkDataBlock control program (Figure 7), which corresponds to a "dispatched" activity that is executed periodically as part of a mission plan.

The control program starts up by initializing the status flag variable 'maintainBCstatus' to the nominal value "sustaining" (line 3). It then continuously asserts the goals of keeping BC A on and BC B off, within the context of an *always* construct (lines 6-9). Note that recoverable failures of BC A (of types "resettable" and "power-cycleable") are handled by the deductive controller without straying from the nominal execution path through the control program: by continuously asserting 'bc_a = on' as a configuration goal, the deductive controller will immediately respond by issuing the "reset" command to BC A or by power cycling it, as appropriate. Should the deductive controller's ME engine ever determine that BC A has failed in an unrecoverable ("broken" mode) or unknown manner, the control program preempts the assertion of 'bc_a = on' and 'bc_b = off' (line 10), and switches over to the redundant backup BC, by asserting the goal 'bc_b = on' (lines 12-14). In the extremely unlikely event that BC B also fails unrecoverably, the control program will terminate, after signaling 'maintainBCstatus = failed' (lines 16-21). Presumably, this would in turn trigger drastic action via hard-wired onboard fault protection, such as entry into safe mode, since communications over the data bus is a mission-critical capability.

Test Plan and Results

In order to be viable, an autonomy system must address a number of operational use cases, including cases derived from experience gained on previous missions such as DS-1. Generally speaking, at a high level these can be categorized as either nominal operations or operations in the presence of faults. It is with these use cases in mind that the preceding models and control programs were developed. Though relatively simple, they cover a large subset of desired features for autonomous spacecraft control. The following sections briefly touch on some of these use cases and how the developed test scenarios have exercised them.

Nominal Operations

Successful monitoring and execution of nominal spacecraft operations are clearly critical, as one typically expects the majority of operations to be conducted in the absence of faults. The nominal operations use case can be further broken down into three sub-categories, which map neatly to elements of the Titan architecture described above. First, an autonomy system must be able to *accept high-level activity goals or procedure invocations and decompose them* into an appropriate sequence of detailed task assignments. This capability is best exemplified in Scenario 1, described above, in which the operator provides only the activity goal to downlink science data. In this scenario, the Titan control sequencer demonstrates the ability to decompose this high-level activity into a valid series of configuration goals for the deductive controller.

A second use case for any autonomy system is to *accept a configuration goal and generate an appropriate sequence of atomic plant commands* that will achieve this goal. More specifically, an ideal autonomy infrastructure will feature the ability to produce both single-step and multi-step reconfiguration sequences. Both Scenarios 1 and 2 highlight the ability of Titan's deductive controller to take configuration goals and produce correct command sequences. While the configuration goals generated in the context of Scenarios 1 and 2 tended to be reachable with a single plant command, multi-step nominal reconfiguration sequences have been demonstrated in Titan using similar scenarios that fall outside the scope of this paper.

Finally, an autonomy system must be able to perform *nominal mode tracking*, or verifying that commands that have been issued to the underlying plant were successfully completed. While both Scenarios 1 and 2 demonstrate the ME engine performing nominal model tracking to some degree, Scenario 2 is the more compelling of the two. In Scenario 2, the "comm" observations gathered from the device components are used to positively reinforce the belief that the bus controller was in fact successfully transitioned from "off" to "on".

Operations in the Presence of Faults

While the general expectation is for the bulk of spacecraft operations to be conducted under nominal operating conditions, the ability to autonomously detect and correct faults is essential, especially in environments where timely ground intervention is impossible. Thus, there exist a number of important use cases related to autonomous operations in the presence of faults.

One core requirement is the *ability to diagnose both single and multiple faults*. In the case of multiple faults, they may occur simultaneously or in succession over time. Titan's ability to diagnose component faults was tested in Scenario 1, where the comm/no-comm status from the devices was used to diagnose bus controller and RT/device faults, and in Scenario 2, where observations were used to diagnose the omni-switch failing to the "stuck-at-B" mode. Another key consideration with respect to fault diagnosis is the management of completely *unanticipated faults*. This ability is represented in the Titan plant models, which all contain a special "unmodeled" or "unknown" state definition. This state definition provides runtime robustness by serving as a fallback for component failure modes not explicitly encoded by the spacecraft engineer. This feature was exercised in a test case of Scenario 1, where a set of plant observations resulted in the engine diagnosing an unmodeled transmitter failure.

Once faults have been detected, the autonomy system must be able to take the necessary action to return to nominal operations. One expectation is that an autonomy system be able to *effect recovery by repairing* the faulty component. This was demonstrated in Scenario 2, where Titan attempts to repair a faulty bus controller by first performing a soft reset, and failing that, by performing a power cycle of the bus controller. As directly repairing a faulty component will not always be possible, an autonomy system must also be able to *take advantage of physical or functional redundancy* to address faults. Again, Titan successfully demonstrated this capability in Scenario 2, where the secondary bus controller was brought online in response to an irreparable failure of the primary bus controller.

Test Visualization

A portion of the scenario verification process also involved a visual presentation of the spacecraft configuration over time as the Titan executive worked to achieve high-level activity goals. For this purpose, the *Helios* visualization tool was employed (Figure 9). Helios is a Java application in the spirit of Stanley,[13] the graphical interface to Livingstone. This tool provides the capability to visualize the ME and MR portions of a deductive controller both graphically, in the form of a schematic display, and textually, in tabular format. The data represented can consist of multiple candidate trajectories over a series of discrete time steps.

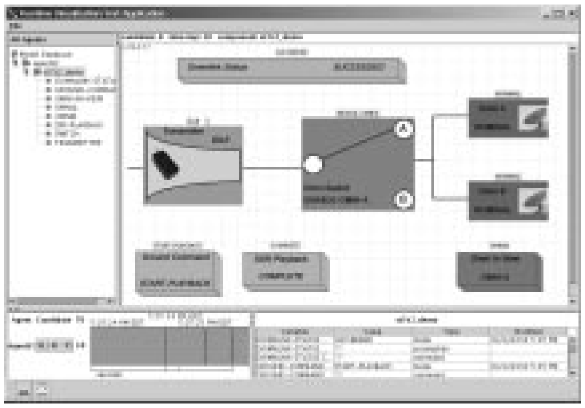

Figure 9. Helios Visualization Tool

CONCLUSIONS

Model-based execution bridges the gap between mission level planning and real-time subsystem commanding in an autonomous system. Titan is a model-based executive that automates diagnosis and commanding of onboard subsystems based on common-sense engineering models of a spacecraft. It provides robustness in the sense-decide-act loop, necessary for operation in the harsh, unpredictable environment of space. This capability can save time and cost in spacecraft operations by decreasing the level of detail needed to command the spacecraft, and decreasing science down-time due to spacecraft safing in response to recoverable faults.

The specification of desired spacecraft behavior in terms of activity goals framed in terms of abstract system state will be an enabling capability for future space missions where high levels of onboard autonomy are required for science optimization, spacecraft safety, or overall cost reduction.

Model-based programming has the potential to fundamentally impact the way future spacecraft systems are conceived, designed, and implemented. In order for this to occur, it will be important to mature the technology with input from a broad range of spacecraft engineering disciplines. The first step in this maturation process is the implementation of control and plant models based on existing spacecraft systems. This effort is currently underway in two mission areas: the DARPA/NASA-funded, MIT-designed SPHERES formation flying and autonomy testbed (to be deployed on the International Space Station in 2003), and NASA's MESSENGER (MErcury Surface, Space ENvironment, GEochemistry and Ranging) Discovery mission. In both of these cases, Titan models and control programs are being developed for existing system designs and mission objectives. In the future, the full benefit of model-based programming will be realized when spacecraft engineers can use tools like Titan and RMPL during mission conceptual and preliminary design, optimizing parameters such as redundancy, observability, and controllability to exploit the full capability of run-time model-based execution.

ACKNOWLEDGEMENTS

Development of the Titan executive was funded in part by the DARPA MoBIES program under contract F33615-00-C-1702 and NASA's Cross Enterprise Technology Development Program under contract NAG2-1466. The modeling and scenario development work was funded as part of NASA's New Millennium Program ST-7 Autonomy Concept Definition Phase Study under NASA/JPL Grant #1232487. The NASA Engineering for Complex Systems Program provided funding for the development of the Helios visualization tool under contract number NAG2-1477.

The authors would like to thank Tony Barrett, Seung Chung, Paul Elliott, Jaime Esper, Steve Marshall, Rob Ragno, Rick Schnurr, Michael Van Steenberg, and Andreas Wehowsky for their contributions to this effort.

REFERENCES

1. D. Bernard, et al., "Spacecraft Autonomy Flight Experience: The DS1 Remote Agent Experiment", *Proceedings of the AIAA Space Technology Conference & Exposition*, Albuquerque, NM, Sept. 28-30, 1999. AIAA-99-4512.
2. M. Ingham, et al., "Autonomous Sequencing and Model-based Fault Protection for Space Interferometry", *Proceedings of the 6th International Symposium on Artificial Intelligence and Robotics & Automation in Space (ISAIRAS-01)*, 2001.
3. C. Goodrich and J. Kurien, "Continuous Measurements and Quantitative Constraints: Challenge Problems for Discrete Modeling Techniques", *Proceedings of the 6th International Symposium on Artificial Intelligence and Robotics & Automation in Space (ISAIRAS-01)*, 2001.
4. B. Williams and P. Nayak, "A Model-based Approach to Reactive Self-Configuring Systems", *Proceedings of the 15th National Conference on Artificial Intelligence (AAAI-98)*, pp. 971-978, 1996.
5. B.C. Williams and M.D. Ingham, "Model-based Programming: Controlling Embedded Systems by Reasoning About Hidden State", to appear, *Proceedings of the 8th International Conference on Principles and Practice of Constraint Programming (CP-02)*, Ithaca, NY, September 2002.
6. R. Schnurr, et al., *Autonomy Technology and Onboard Processing Study – Space Technology 7-Autonomy and Autonomy Pilot Project*, Study report submitted to the New Millennium Program and the Office of Space Science by NASA Goddard Space Flight Center, February 8, 2002, in press.
7. B. Pell, et al., "A Hybrid Procedural/Deductive Executive For Autonomous Spacecraft", In *Proceedings of the 2nd International Conference on Autonomous Agents (Agents '98)*, Minneapolis, MI, May 1998.
8. S. Chien, et al., "ASPEN - Automating Space Mission Operations using Automated Planning and Scheduling", *Proceedings of Space Operations 2000 Conference (SpaceOps 2000)*, Toulouse, France, June 2000.
9. A.K. Jonsson, et al., "Planning in Interplanetary Space: Theory and Practice", in *Proceedings of the 5th International Conference on Artificial Intelligence Planning Systems (AIPS-00)*, Breckenridge, CO, 2000.
10. P. Kim, B.C. Williams, and M. Abramson, "Executing Reactive, Model-based Programs Through Graph-based Temporal Planning", *Proceedings of the 17th International Joint Conference on Artificial Intelligence (IJCAI-01)*, Seattle, WA, 2001.
11. B. Williams and P. Nayak, "A Reactive Planner for a Model-based Executive", *Proceedings of the 15th International Joint Conference on Artificial Intelligence (IJCAI-97)*, Nagoya, Japan, August 1997.
12. B.C. Williams and R.J. Ragno, "Conflict-Directed A* and its Role in Model-based Embedded Systems", to appear, *Journal of Discrete Applied Mathematics*.
13. K. Rajan, et al., "Ground Tools for the 21st Century", *Proceedings of IEEE Aerospace Conference*, Big Sky, MT, 2000.