An implemented architecture integrating onboard planning, scheduling, execution, diagnosis, monitoring and control for autonomous spacecraft

Barney Pell Douglas E. Bernard Steve A. Chien Erann Gat
Nicola Muscettola P. Pandurang Nayak Michael D. Wagner
Brian C. Williams

January 17, 1996

Abstract

NASA has recently announced the New Millennium Program (NMP) to develop "faster, better, cheaper" spacecraft in order to establish a "virtual presence" in space. A crucial element in achieving this vision is onboard spacecraft autonomy, requiring us to automate functions which have traditionally been achieved on ground by humans. These include planning activities, sequencing spacecraft actions, tracking spacecraft state, ensuring correct functioning, recovering in cases of failure and reconfiguring hardware.

In response to these challenging requirements, we analyzed the spacecraft domain to determine its unique properties and developed an architecture which provided the required functionality. This architecture integrates traditional real-time monitoring and control with constraint-based planning and scheduling, robust multi-threaded execution, and model-based diagnosis and reconfiguration.

In a five month effort we successfully demonstrated this implemented architecture in the context of an autonomous insertion of a simulated spacecraft into orbit around Saturn, trading off science and engineering goals, and achieving the mission goals in the face of any single point of hardware failure. This scenario turned out to be among the most complex handled by each of the component technologies. As a result of this success, the integrated architecture has been selected to control the first NMP flight, Deep Space One, in 1998. It will be the first AI system to autonomously control an actual spacecraft.

keywords: AI architectures, applications, robotics

electronic id: A135

1 Introduction

The future of space exploration calls for establishing a "virtual presence" in space. This will be reached with a large number of smart, cheap spacecraft carrying on missions as ambitious as robotic rovers, balloons for extended atmospheric explorations and robotic submarines. Several new technologies need to be demonstrate to reach this goal, and one of the most crucial is certainly on-board spacecraft autonomy.

In the traditional approach to spacecraft operations humans carry out on the ground a large number of functions including planning activities, sequencing spacecraft actions, tracking the spacecraft's internal hardware state, ensuring correct functioning, recovering in cases of failure, and subsequently working around faulty subsystems. This approach will not be viable anymore in the future due to (a) round trip light time communication delays which make joysticking a deep space mission impossible and (b) a desire to limit the operations team and deep space network (DSN) costs.

In the new new model of operations, the scientists will communicate high-level science goals directly to the spacecraft. The spacecraft will then perform its own science planning and scheduling, translate those schedules into sequences, verify that they will not damage the spacecraft, and ultimately execute them without routine human intervention. In the case of error recovery, the spacecraft would have to understand the impact of the error on its previously planned sequence and then reschedule in light of the new information.

This work has been carried out within NASA's New Millennium Program (NMP), a series of aggressive technology-demonstration space missions. In order to assess and demonstrate the applicability of AI technology to spacecraft autonomy, the NMP formed a team combining AI researchers with some of the best spacecraft engineers with the objective of developing and demonstrating an architecture integrating AI tools with traditional spacecraft control. The challenge was to demonstrate complete autonomous operations in a very challenging context: insertion of a Cassini-like spacecraft into orbit around Saturn, trading off science and engineering goals, and achieving the mission in the face of any single point of hardware failure. The scenario was proposed by experienced spacecraft engineers who had participated in several previous planetary missions. Although simplified, it still contains the most important constraints and sources of complexities of a real mission, making it the most difficult challenge in the context of the most complicated mission phase of the most advanced spacecraft to date. Furthermore, the demonstration had to be accomplished in the very short time frame of about 6 months.

The unique requirements of this domain led us to the New Millennium Remote Agent (NMRA) architecture which integrates traditional real-time monitoring and control with (a) constraint-based planning and scheduling, to ensure achievement of long-term mission objectives and effectively manage allocation of scarce system resources; (b) robust multi-threaded execution, to reliably execute planned sequences under conditions of uncertainty, to rapidly respond to unexpected events such as component failures, and to manage concurrent real-time activities; and (c) model-based diagnosis, to confirm successful plan execution and to infer the health of all system components based on inherently limited sensor information.

The New Millennium Remote Agent (NMRA) architecture was successfully demonstrated on the SOI scenario last October. The successful demonstration resulted in the inclusion of NMRA in the flight software of the first NMP mission, Deep Space 1 (DS-1), which is scheduled to launch in mid-1998. This will be the first AI system to autonomously control an actual spacecraft.

2 Scenario

2.1 Introduction

A simplified Saturn Orbit Insertion (SOI) scenario was used to define the requirements on the technologies and the level of detail needed in modeling the spacecraft. It included goals and constraints and an example sequence to satisfy the goals and constraints. The challenge to the autonomous system was not to duplicate this sequence, but rather to plan and execute tasks in such a manner that all constraints were satisfied.

The scenario also included failure scenarios. The failure recovery requirements are as follows:

1. Achieve the mission goals even in the event of any single point hardware failure.
2. Consider the Saturn Orbit Insertion burn a special event that, for robustness, requires that all critical subsystems operate in their highest reliability modes.
3. Although multiple independent simultaneous failures are not considered credible, multiple sequential failures–spaced far enough apart to allow recovery of one before considering the next–are considered credible and must be accommodated.

2.1.1 Goals

The following goals define the SOI scenario:

- Use the main engine to insert the spacecraft into Saturn Orbit
- Acquire and return science images of Saturn during approach
- Acquire and return science images of Saturn's rings near closest approach
- Assure that the camera is protected from ring particles during ring-plane crossing

2.1.2 Constraints

The models of the spacecraft as understood by the planner form the context for achieving the above goals. These models constrain the choices that the planner may make, force certain tasks to be ordered, and force the addition of tasks to allow the goals to be achieved. For the SOI scenario, the following constraints significantly affect the resulting plan:

- Available spacecraft electrical power is limited; each operating mode of each assembly requires a predefined power allocation.
- Available science data storage is limited; there is not enough room to accommodate both the Saturn approach and Saturn ring images simultaneously
- Only one spacecraft pointing direction may be commanded at a time. This couples the science imaging activity, the orbit change activity, the Earth communication activity, and the ring safety activity since all require some spacecraft axis to be pointed in a particular direction (e.g., antenna toward earth).
- A main engine burn requires several preparatory steps prior to engine ignition.

2.1.3 One Possible Sequence

One possible sequence of events that meets the goals and constraints is shown below. Other sequences are possible, and changes may be made during execution such that the specific task order described here is not followed, but all constraints between tasks are satisfied.

2.1.4 Saturn Orbit Insertion Scenario Details

The scenario begins one day before initial Saturn periapsis.

A plan is then generated on-board based on current on-board information about the state of the spacecraft, the spacecraft trajectory with respect to Saturn, the goals for the Saturn orbit insertion mission phase, and the system constraints.

Ground controllers desire to know about the success of certain risky activities, such as the firing of pyrotechnic devices early enough to take action if failures occur. This forces certain activities to be scheduled early, followed by communication of the results to the ground controllers on Earth.

Science images are desired of Saturn initial approach and of the rings during closest approach. Limited data recorder space means that the plan should include the recorder download after the approach imaging and before the ring imaging.

Power is a limited resource and engine ignition for the SOI burn occurs when power is the tightest. Non-essential equipment (e.g., science instruments and reaction wheels) must be powered off prior to engine ignition.

Some devices need to be warmed up prior to use. Each must be turned on early enough to assure availability when needed.

For the critical SOI mission phase, backup units are also warmed up and ready to go.

The main engine is prepared for use by powering on its electronics, opening latch valves, and pre-aiming the gimbaled engine. These activities are scheduled early enough that failures allow time to switch to the backup engine.

During SOI preparation and science collection, the spacecraft crosses the Saturn ring plane and must go to an attitude that shields the camera from ring particles.

The spacecraft turns to the burn attitude, main engine ignition occurs, and the spacecraft is inserted into Saturn orbit.

After the burn, the spacecraft is returned to a safe state

The orbit insertion burn is scheduled to end at periapsis so that science observations may take advantage of the closest approach viewing.

The ring-plane images are downlinked to the Earth as soon as possible.

After transmission of science and engineering data to the ground, the scenario is complete.

The following failure scenarios also had to be handled successfully:

The main engine overheats during the burn. An overheated engine can damage the rest of the spacecraft, so a reflex response is needed to shut the burn down upon detection. The backup engine will then be used on the next burn attempt. This requires re-planning with burn restart time scheduled for when all propulsion equipment has cooled down sufficiently. The duration of the new burn must be adjusted based on the amount of burn accomplished in the first attempt.

A gyroscope fails to give data. Since the backup gyroscope is on and warmed up, a simple switch is performed while the burn continues without interruption.

3 Domain and Requirements

The spacecraft domain places a number of requirements on the software architecture that differentiates it from domains considered by other researchers. There are three major properties of the domain that drove the architecture design.

First, a spacecraft must be able to carry on *autonomous operations for long periods of time* with no human interaction. This requirement stems from (a) round trip light time communication delays which make joysticking a deep space mission impossible and (b) a desire to limit the operations team and deep space network (DSN) costs.

The requirement for autonomous operations over long periods is further complicated by two additional features of the domain–*tight resource constraints* and *hard deadlines*. A spacecraft uses various resources, including obvious ones like fuel and electrical power, and less obvious ones like the number of times a battery can be reliably discharged and recharged. Some of these resources are renewable but most of them are not. Hence, autonomous operations requires significant emphasis on the careful utilization of non-renewable resources and on planning for the replacement of renewable resources before they run dangerously low. Spacecraft operations are also characterized by the presence of hard deadlines due to the fact that the efficiency of orbit change maneuvers is an extremely strong function of the location of the spacecfraft in its orbit–which is a function of time. For example, the time at which SOI must be achieved is constrained to lie within a two hour window. Sophisticated planning and scheduling system should be used to ensure th previous requirement.

The second central requirement of spacecraft operation is *high reliability*. Since a spacecraft is very expensive and often unique, it is essential that it achieve its mission with a very high level of reliability. Part of this high reliability is achieved through the use of very reliable hardware. However, the harsh environment of space or the inability to test in all flight conditions can still cause unexpected hardware failures, so that the software architecture is required to compensate for such contingencies. This requirement dictates the use of an executive and elaborate system-level fault protection that can rapidly react to contingencies by retrying failed actions, reconfiguring spacecraft subsystems, or safing the spacecraft to prevent further, potentially irretrievable, damage. Of equal danger are catastrophic software bugs, often introduced through a mismatch of spacecraft models in the heads of different software engineers. This requirement dictates the need to maximize the use of a consistent model shared between the different executive functions.

The requirement of high reliability is further complicated by the fact that there is *limited observability* into the spacecraft's state due to the availability of a limited number of sensors. The addition of sensors implies added mass[1], power, cabling, and up front engineering time and effort. Each sensor must add clear value to the mission to be justified for inclusion. Furthermore, sensors are typically no more reliable than the associated spacecraft hardware, making it that much more difficult to deduce the true state of the spacecraft hardware. These requirements dictate the use of sophisticated model-based diagnosis methods for identifying the true state of the spacecraft hardware. These methods predict unobservable state variables using a spacecraft model, and can effectively handle sensor failures. In addition these diagnostic methods must be augmented with sophisticated model-based control methods that help the executive to reconfigure hardware in view of failure knowledge and to predict the consequences of these actions.

The third central requirement of spacecraft operation is that of *concurrent activity*. The spacecraft has a number of different subsystems, all of which operate concurrently. Hence, reasoning about the spacecraft needs to reflect its concurrent nature. In particular, the planner/scheduler needs to be able to schedule concurrent activities in different parts of the spacecraft, including constraints between concurrent activities. The executive needs to have concurrent threads active to handle concurrent commands to different parts of the spacecraft. The model-based diagnosis and reconfiguration system needs to handle concurrent changes in the spacecraft state, either due to scheduled events or due to failures.

4 Architecture Overview

In the architecture autonomous operations is achieved through the cooperation of 5 distinct components (Figure 1).

Continuous autonomous operation is achieved by the repetition of the following cycle.

1. Retrieve high level goals from the mission's goals database. In the actual mission, goals can be known at the beginning of the mission, put into the database by communication from ground mission control or can originate from the operations of spacecraft subsystems (e.g., "take more pictures of star fields to estimate position and velocity of the spacecraft").

[1]In a spacecraft, mass directly translates to the cost of launch and the cost of carrying extra fuel to achieve all mission maneuvers.

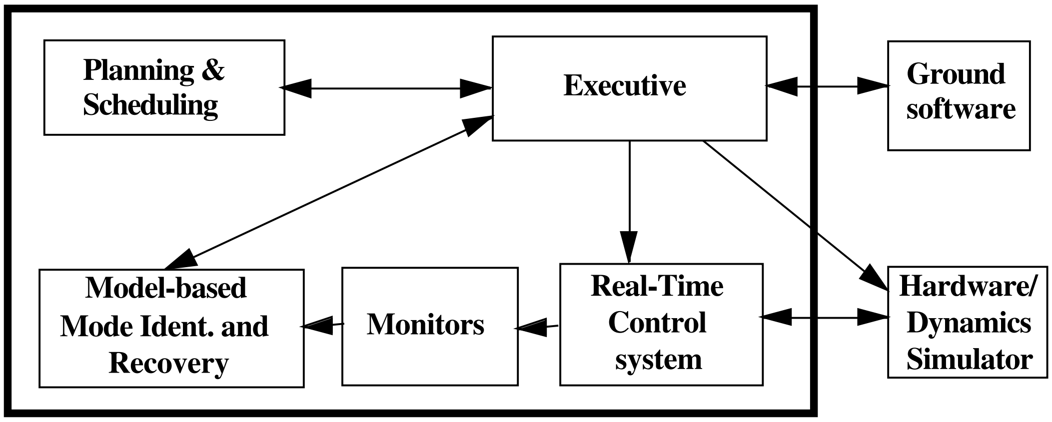

Figure 1: NMRA architecture

2. Ask the *planner/scheduler* to generate a schedule. The planner receives the goals, the scheduling horizon, i.e., the time interval that the schedule needs to cover, and an initial state, i.e., the state of all relevant spacecraft subsistems at the beginning of the scheduling horizon. The resulting schedule is represented as a set of *tokens* placed on various *state variable time lines*, with temporal constraints between tokens.

3. Send the new schedule generated by the planner to the *executive*. The executive will continue executing its current schedule and start executing the new schedule when the clock reaches the beginning of the new scheduling horizon. The executive translates the abstract tokens contained in the schedule into a sequence of lower level spacecraft commands that correctly implement the tokens and the constraints between tokens. It then executes these commands, making sure that the commands succeed and either retrying failed commands or generating an alternate low level command sequence that achieve the token. Hard command execution failures may require the modification of the schedule in which case the executive will coordinate the actions needed to keep the spacecraft in a "safe state" and request the generation of a new schedule from the planner.

4. Repeat the cycle from step 1 when one of the following conditions apply:

   (a) Execution (real) time has reached the end of the scheduling horizon minus the estimated time needed for the planner to generate a schedule for the following scheduling horizon;

   (b) The executive has requested a new schedule as a result of a hard failure.

Schedule execution is achieved through the cooperation of the the executive and the other three architectural layers. The executive reasons about spacecraft state in terms of a set component modes. The *mode identification* (MI) layer is responsible for providing this level of abstraction to the executive. MI takes as input the executive command sequence and observations from sensors to identify the current mode (nominal or failed) of each spacecraft component. The *monitoring* layer takes the raw sensor data stream, and discretizes it to the abstract level required by MI. Finally, the *control and real-time system* layer takes commands from the executive and provides the actual control of the low level state of the spacecraft. It is responsible for providing the low level sensor data stream to the monitors.

The 4 lower layers are always active and in concurrent execution. This ensures the high reliability required by the domain. The planner/scheduler is the only component that is activated as a "batch process" and dies after a new schedule has been generated.

Monitoring and control follow traditional approaches to spacecraft software and will not be discussed here. In the following we will concentrate on the other modules.

4.1 Planner

The goal of the planner/scheduler is to generate a set of synchronized high-level commands that once executed will achieve mission goals.

Particularly in the spacecraft domain planning and scheduling aspects of the problem need to be tightly integrated. Clearly the planner needs to recursively select and schedule appropriate activities to achieve mission goals and any other subgoals generated by these activities. It also needs to synchronize activities and allocating global resources over time (e.g., power and data storage capacity). However in this domain (but this is also true in general) subgoals may be generated also due to limited availability of resources over time. For example, in a mission it would be preferrable to keep scientific intruments on as long as possible (to maximize the amount of science gathered). However limited power availability may force a temporary instrument shut-down when other more mission critical subsytems need to be functioning. In this case the allocation of power to critical subsystems (the main result of a scheduling step) generates the subgoal "instrument must be off" (which requires the application of a planning step). Considering simultaneusly the consequences of planning and scheduling steps enables a planning algorithm to exert more control on the order in which decisions are made and to therefore keep search complexity under control.

Besides activities, the planner must also "schedule" the occurrence states and conditions that need to be monitored to ensure that high level spacecraft conditions are correct for goals (such as spacecraft pointing states, spacecraft acceleration and stability requirements, etc.). These states can also consume resources and have finite durations.

The planner used in the NMRA architecture consists of a heuristic search engine operating on a temporal database. The search engine posts constraints on the basis of external goals or constraint templates stored in a model of the spacecraft. Using an iterative sampling approach, the planner also tries to heuristically improve on certain aspects of schedule quality, although it does not guarantee even local optimality along this metric. The temporal database and the facilities for defining and accessing model information during search are provided by the HSTS system (Muscettola 1994).

The domain model contains an explicit declaration of the spacecraft subsystems on which an activity or a state will occur. In the temporal database each subsystem has an associated timeline on which the planner inserts activities and states and resolves resource allocation conflicts. The model also contains the declaration of duration constraint and of templates of temporal constraints between activities and states. Such constraints have to be satisfied by any schedule stored in the temporal database for it to be consistent with the physics of the domain. Temporal constraint templates absolve the role of generalized planning operators and are defined for any activity or state in the domain. The temporal database also provides constraint propagation services to verify the global consistency of the constraints posted so far.

The constraint template in Figure 2 describes the 6 conditions needed for an engine burn to initiate correctly (activity *Engine_Burn_Ignition* scheduled on the *(Engine Op_State)* timeline). Constraint 6 represents a request for power that increases the level of *Power_Used* on the timeline *(NewMaap_Power_Mgmt Power)* of an amount returned by the Lisp function call *(compute-power 'Engine_Burn_Ignition)*. Explicit declaration of function calls in the model such as the one above provides the means for the planner to invoke "expert" modules expert modules to provide narrow but deep levels of expertise in the computation of various parameters such as durations or temperature and power levels.

4.2 Hybrid Executive

The executive is responsible for performing runtime management of all system activities. The executive's functions include process synchronization, process dependency management, hardware reconfiguration and runtime resource management, and the execution of fault recovery procedures. The executive invokes the planner and mode identification components to help it perform these functions. The executive also controls the low-level control software by setting its modes and supplying parameters and by responding to monitored events. The executive thus performs similar functions to a traditional operating system. The main difference is that when unexpected contingencies occur, a traditional operating system can only issue a report and abort the offending process, relying on user intervention to recover from the problem. Our executive must be able to take corrective action automatically, for example in order to meet a tight orbital insertion window. Our approach involved the development of a hybrid executive that shares execution responsibilities between a classical reactive execution system, RAPS (Firby 1978) and a novel model-based reconfiguration system, called Livingstone.

RAPS provides a specialized representation language for describing context-dependent contingent response procedures, with an event-driven execution semantics. The language ensures reactivity, is natural for decomposing tasks and corresponding methods, and makes it easy to express monitoring and contingent action schemas. Its runtime system then manages the reactive exploration of a space of alternative action by searching through a space of task decompositions. RAPS encourages a close adherance to a reactive programming principle of limiting deductions within the sense-act loop to that of constructing task decompositions using a limited form of matching. This ensures quick response time, which is essential to the survival of the spacecraft. Nevertheless it places a burden on the programmar of

```
(Define_Compatibility
  ((Engine Op_State) (Engine_Burn_Ignition))
  :compatibility_spec
  (AND
   ;; 1. The pressure in the engine tanks must be good during ignition
   (contained_by ((Engine_Tanks Pressure) (Engine_Tanks_Pressure_Good)))

   ;; 2. The Engine must have been finished late burn preparation
   (met_by ((Engine Op_State) (Engine_Burn_Late_Prep)))

   ;; 3. The Engine goes into sustained burn state next
   (meets ((Engine Op_State) (Engine_Burn)))

   ;; 4. The injector temperature must be in range at start of burn
   (contained_by ((Engine_Injector Temp) (Temperature(Ready))))

   ;; 5. Needs VDECU on
   (contained_by ((VDECU Op_State) (VDECU_On)))

   ;; 6. The following amount of Power will be consumed
   (equal ((NewMaap_Power_Mgmt Power)
           ( + (Lisp  (compute-power 'Engine_Burn_Ignition) )
               Power_Used)))))

(Define_Duration_Spec
  ((Engine Op_State) (Engine_Burn_Ignition))

   ;; minimum duration
   (Lisp  (compute-duration 'Engine_Burn_Ignition :minimum) )
   ;; maximu duration
   (Lisp  (compute-duration 'Engine_Burn_Ignition :maximum) )
  )
```

Figure 2: Constraints on the *Engine_Burn_Ignition* activity

deducing *a priori* the consequences of failures and contingencies. This is exacerbated by subtle hardware interactions, multiple and unmodeled failures, the mixture of interactions between computation, electronics and hydraulic subsystems, and limited observability due to sensor costs.

The model-based reconfiguration system, Livingstone, complements these reactive capabilities by providing a set of deductive capabilities along the sense-act loop that operate on a single, compositional model. These models permit significant on the fly deduction of system wide interactions, used to process new sensor information or to evaluate the effects of alternate recovery actions. Yet Livingstone respects the intent of reactive systems, using propositional deductive capabilities coupled to anytime algorithms that have proven exceptionally efficient in the model-based diagnosis of causal systems. Hence Livingstone is able to reason reactively from knowledge of failure, through the models, to optimal actions that restablish the planner's primitive goals while obviating the failures' effects.

Nevertheless, the assurance of fast inference is achieved through strong restrictions on the representation used for possible recovery actions and even more severe limitations on the way in which these actions are combined. If reactivity is to be preserved, then the only alternative is for a programmar or deductive system to script these complex actions before the fact. Hence RAPS provides a natural complement to Livingstone's deductive capabilities. For example, with respect to recovery, Livingstone provides a service for selecting, composing together and deducing the effects of basic actions, in light of failure knowledge. Meanwhile RAPS provides powerful capabilities for elaborating and interleaving these basic actions into more complex sequences, which in turn may be further evaluated through Livingstone's deductive capabilities.

4.3 Mode identification

The *mode identification* (MI) layer of the NMRA architecture is responsible for identifying the current operating or failure mode of each component in the spacecraft. MI is the sensing component of Livingstone's model-based reconfiguration capability, and provides a layer of abstraction to the executive: it allows the executive to reason about the state of the spacecraft in terms of component modes, rather than in terms of low level sensor values.

MI provides a variety of functions within the overall architecture. These include:

- Mode confirmation: Provide confirmation to the executive that a particular spacecraft command has completed successfully.

- Anomaly detection: Identify observed spacecraft behavior that is inconsistent with its expected behavior.

- Fault isolation and diagnosis: Identify components whose failures explain detected anomalies. In cases where models of component failure exist, identify the particular failure modes of components that explain anomalies.

- Token tracking: Monitor the state of planner tokens, allowing the executive to monitor plan execution.

MI uses algorithms adapted from model-based diagnosis (de Kleer & Williams 1987; 1989) to provide the above functions. The key idea underlying model-based diagnosis is that the current state of the spacecraft can be described by a combination of component modes only if the set of models associated with these modes is consistent with the observed sensor values. Following (de Kleer & Williams 1989), MI uses a conflict directed best-first search to find the most likely combination of component modes consistent with the observations. Note that this methodology is independent of the actual set of available sensors. Furthermore, it does not require that all aspects of the spacecraft state are directly observable, providing an elegant solution to the problem of limited observability discussed in Section 3.

The use of model-based diagnosis algorithms immediately provides MI with a number of additional features. First, the search algorithms are sound and complete, providing a guarantee of coverage with respect to the models used. Second, the model building methodology is modular, which simplifies model construction and maintenance, and supports reuse. Third, the algorithms extend smoothly to handling multiple faults. Fourth, while the algorithms do not require explicit fault models for each component, they can easily exploit available fault models to find likely failures.

MI extends the basic ideas of model-based diagnosis by modeling each component as a finite state machine, and the whole spacecraft as a set of concurrent, synchronous state machines. Modeling components as finite state machines allows MI to effectively track state changes resulting from executive commands. Modeling the spacecraft as a concurrent machine allows MI to effectively track concurrent state changes caused either by executive commands or component failures.

Another important feature of MI is that it models the behavior of each component mode using abstract, or qualitative, models (Weld & de Kleer 1990; de Kleer & Williams 1991). These abstract models are encoded as a set of propositional clauses, allowing the use of efficient unit propagation for behavior prediction. In addition to supporting efficient behavior prediction, abstract models are much easier to acquire than detailed quantitative engineering models, and yield more robust predictions since small changes in the underlying parameters do not affect the abstract behavior of the spacecraft. Spacecraft modes are a symbolic abstraction of nondiscrete sensor values and are synthesized by the monitoring module.

Finally, Livingstone uses a single model to perform all of MI's functions, also used for the executive functions of model-based recovery and reconfiguration. It also uses the kernel algorithm, generalized from diagnosis, to perform all of these MI and executive functions. The combination of a small kernel with a single model, and the process of exercising these through multiple uses, contributes significantly to the robustness of the complete system.

5 Implementation

The implemented NMRA architecture succesfully demonstrated planning of a nominal scenario, concurrent execution and monitoring, fault isolation, recovery and re-planning on a simulation of the simplified Csssini SOI scenario.

The planner modelled the domain with 22 parallel timelines and 52 distinct temporal constraint templates. Each template included an average of 3 temporal constraints of which an average of 1.4 constraints synchronized different timelines. The resulting schedule for the nominal scenario included 200 distinct time intervals; a schedule generated after re-planning due to engine burn interruption included 123 time intervals. The planner generated these schedules exploring less than 500 search states in an elapsed time of less than 15 minutes on a Sparc10. Considering the computational resources available in the DS-1 mission and the background nature of the planning process, this speed is acceptable with respect to the performance needed for DS-1.

The executive contained 100 raps with an average of 2.7 steps per raps. The nominal schedule was translated into a task net with 465 steps, making it the biggest RAP to date. The executive interacted with the underlying control loops which operated at a cycle frequency of 4 Hz. This performace level is actually higher than that needed to meet the requirements of the DS-1 mission.

The SOI model for the mode identification and recovery system included 80 spacecraft components with an average of 3.5 modes per component. The structure and dynamics of the domain was captured by 3424 propositions and 11101 clauses. In spite of the very large size of the model, the conflict-centered algorithms permitted fast fault isolation and determination of recovery actions. Fault isolation took between 4 and 16 search steps (1.1 to 5.5 seconds on a SPARC-5) with an average of 7 steps (2.2 seconds). Recovery took between 4 and 20 steps (1.6 to 6.1 seconds) with an average of 9.3 steps (3.1 seconds).

6 Discussion

Important aspects of our architecture can be viewed relative to heterogeneous architectures and with constrast to the mobile robot domain.

6.1 Heterogeneous Knowledge Representation

The research approach to an architecture for autonomy is usually to seek a unified system based on a uniform representational and computational framework. While this is a very important goal, often the complexity of a real-world domain forces researchers to compromise on complete autonomy or to address simpler domains and applications. In our case the challenge was to achieve complete autonomy for a very complex domain in a limited amount of time. Therefore we chose from the outset to use state-of-the-art, general-purpose components that had been applied to solving isolated problems in the domain. The main architectural challenge was therefore to integrate these components. The main source of difficulty here was that our computational engines all require different representations. This heterogeneity has both benefits and difficulties.

One benefit of having each engine look at the spacecraft from a different perspective is that the heterogeneous knowledge acquisition process aids in attaining *coverage and completeness*. Each new perspective on a subsytem potentially increases the understanding, and hence improves the modeling, for each of the other components which also represent knowledge of that subsystem. Another benefit is *redundancy*, where overlapping models enable one component to compensate for restrictions in the representation of another component. This is particulary true for overlapping responsibility in the hybrid executive. A third benefit is *task specialization*, in which each component is optimized for solving certain kinds of tasks. This means that we can use each component to solve problems for which it is well suited, rather than requiring one component to solve all problems (a similar point is made by R. P. Bonasso & Slack (1995)).

An important example of representational differences that we found was between the planner/scheduler and the hybrid execution system. In NMRA the planner is concerned with activities at a high-level of abstraction which encapsulates a detailed sequence of executive-level commands. A fundamental objective for the planner is to allocate resources to the high-level activities so as to provide a time and resource envelope that will ensure correctness of execution for each executive-level detailed sequence. An interval based representation is eminently suitable for this purpose. From this perspective the planner does not really need to know if a time interval pertains to an activity or a state. However, this knowledge is instead crucial to ensure a correct execution. The executive is eminently interested in the occurrence of event, the transition between time intervals in the planner's perspective. To generate the appropriate commands and set up the appropriate sensor monitors, the executive needs to know if an event is controllable (the executive needs to send a command), observable (the executive expect sensory information) or neither (the executive can deduce information on the state on the basis of the domain model). Our approach localizes such distinctions to the executive's knowledge representation. This frees the planner to reason efficiently about intervals, and enables us to move responsibility flexibly between other architectural components (for example, let the control tasks handle an activity which was formerly decomposed by the executive, or vice-versa) without having to modify the planner's models.

While heterogeneous representations have a number of benefits, they also raise some difficulties. Most significant of these are the possibility for models to diverge rather than converge, and the need to duplicate knowledge representation efforts. We have made some progress on this front by heading toward a more unified representation of some modeled properties. First, the unified modeling for MI/MR in Livingstone (see Section 4.3) has proven to be extremely useful. Second, we use code generation techniques to translate some modeled properties, such as device power requirements, into the different representations used for each computational engine. Ideally, we would like to head toward a single representation of the spacecraft (the *one true model*, a holy grail of AI), but we intend to do so always generalizing from powerful models capable of handling the complexities of our real-world domain.

6.2 Differences with the Mobot Domain

Many of the AI autonomy architectures have been developed with respect to mobile robots (mobots). Two differences in particular are the role of *perception* and *failure handling* in the two domains.

Many of the problems of perception common in mobile robot architectures were not significant in our domain. NMRA is focused on the spacecraft's state, and sensing the state of a synthetic artifact is much easier than sensing and understanding a complex natural environment. Furthermore, only limited aspects of the relationship of the spacecraft to its environment were sensed using sophisticated sensors, e.g., spacecraft acceleration, spacecraft angular velocity, sun position. Results from such sensors are easy to understand and incorporate into the model of the spacecraft's state.

Second, there are important differences in the structure of unexpected contingencies between the spacecraft domain and the mobile robot domain. The major difference is that there are almost no serendipitous contingencies on spacecraft. Because spacecraft are carefully designed to perform a narrow, specific mission, and any deviation is considered a failure. By contrast, multiple outcomes of actions and unexpected contingencies for mobots are often difficult to dichotomize into success and failure; mobots can sometimes achieve their goals by performing random actions. This distinction is manifested in the design of the RAP language, which recognizes failure of a plan step, but does not provide a mechanism for failure recovery per se. Instead, failure recovery procedures must be written like any other method, to be triggered on the result and context of the failure rather than the failure itself.

Moreover, mobots are typically concerned with failures in the interaction between robot and environment. These failures are typically intermittent. In the case of spacecraft, a permanent hardware failure will not go away even if the system recovers this time. Having now limited capabilities, the agent must plan and execute behavior with new constraints in mind, and make future inferences relative to the new system state. This raises a need for a system-level approach to fault protection, which ultimately resulted in the important role of Livingstone and in several architectural requirements to support replanning in the case of failures.

7 Related Work

The New Millennium Remote Agent (NMRA) architecture is closely related to the 3T (three-tier) architecture described in (R. P. Bonasso & Slack 1996). The 3T architecture consists of a deliberative component and a real-time control component connected by a reactive conditional sequencer. We and Bonnasso both use RAPS (Firby 1978) as our sequencer, although we are developing a new sequencer which is more closely tailored to the demands of the spacecraft environment (Gat 1996). Our deliberator is a traditional AI planner based on the HSTS temporal database (Muscettola 1994), and our control component is a traditional spacecraft attitude control system (Hackney, Bernard, & Rasmussen 1993). We also add an architectural component explicitly dedicated to world modelling (the mode identifier), and distinguish between control and monitoring. In contrast to the system described by Bonnasso, the prime mover in our system is the RAP sequencer, not the planner. The planner is viewed as a service invoked and controlled by the sequencer. This is necessary because computation is a limited resource (due to the hard time constraints) and so the relatively expensive operation of the planner must be carefully controlled. In this respect, our architecture follows the design of the ATLANTIS architecture (Gat 1992).

The current state of the art in spacecraft autonomy is represented by the attitude and articulation control subsystem (AACS) on the Cassini spacecraft (Brown, Bernard, & Rasmussen 1995; Hackney, Bernard, & Rasmussen 1993) (which supplied the Saturn Orbit Insertion scenario used in our prototype). The autonomy capabilities of Cassini include context-dependent command handling, resource management and fault protection. Planning is a ground (rather than on-board) function and on-board replanning is limited to a couple of predefined contingencies. An extensive set of fault monitors is used to filter measurements and warn the system of both unacceptable and off-nominal behavior. Fault diagnosis and recovery are rule-based. That is, for every possible fault or set of faults, the monitor states leading to a particular diagnosis are explicitly encoded into rules. Likewise, the fault responses for each diagnosis are explicitly encoded by hand. Robustness is achieved in difficult-to-diagnose situations by setting the system to a simple, known state from which capabilities are added incrementally until full capability is achieved or the fault is unambiguously identified. The NMRA architecture uses a model-based fault diagnosis system, adds an on-board planner, and greatly enhances the capabilities of the on-board sequencer, resulting in a dramatic leap ahead in autonomy capability.

Ahmed, Aljabri, & Eldred (1994) have also worked on architecture for autonomous spacecraft. Their architecture integrates planning and execution, using TCA (Simmons 1990) as a sequencing mechanism. However, they focused only on a subset of the problem, that of autonomous maneuver planning, which will be incorporated into our work as part of the DS-1 mission.

Among the many general-purpose autonomy architectures is Guardian (Hayes-Roth 1995), a two-layer architecture which has been used for medical monitoring of intensive care patients. Like the spacecraft domain, intensive care has hard real-time deadlines imposed by the environment and operational criticality. One notable feature of the Guardian architecture is its ability to dynamically change the amount of computational resources being devoted to its various components. The NMRA architecture also has this ability, but the approaches are quite different. Guardian manages computational resources by changing the rates at which messages are sent to the various parts of the system. The NMRA architecture manages computational resources by giving the executive control over deliberative processes, which are managed according to the knowledge encoded in the RAPs.

SOAR (Laird, Newell, & Rosenbloom 1987) is an architecture based on a general-purpose search mechanism and a learning mechanism that compiles the results of past searches for fast response in the future. SOAR has been used to control flight simulators, a domain which also has hard real-time constraints and operational criticality (Tambe *et al.* 1995). CIRCA (Musliner, Durfee, & Shin 1993) is an architecture that uses a slow AI component to provide guidance to a real-time scheduler that guarantees hard real-time response when possible within the constraints. Noreils & Chatila (1995) describes a mobile robot control architecture that combines planning, execution, monitoring, and contingency recovery. Cypress is an architecture which combines a planning and an execution system (SIPE-II and PRS (Georgeff & Lansky 1987)) using a common representation called ACTS (Wilkins & Myers 1995). The main difference between Cypress and our system is our use of an interval-based planner rather than an operator-based planner.

8 Conclusions and Future Work

This paper has described NMRA, an implemented architecture for autonomous spacecraft. The architecture was driven by a careful analysis of the spacecraft domain, and integrates traditional real-time monitoring and control with constraint-based planning and scheduling, robust multi-threaded execution, and model-based diagnosis and reconfiguration. The implemented architecture was successfully demonstrated on an extremely challenging spacecraft autonomy scenario. As a result, the architecture will control the first flight of NASA's New Millennium Program (NMP). The spacecraft, NMP Deep Space One (DS-1), will launch in 1998 and will autonomously cruise to and fly-by an asteroid and a comet. This will be the first AI system to autonomously control an actual spacecraft.

Our immediate work for DS-1 consists mainly in acquiring and validating models of the DS-1 spacecraft and in eliciting and addressing mission requirements. To make this possible, we are working on developing better tools for sharing models across the different heterogeneous architectural components, and for model verification and validation.

Longer term, we see at least three major areas of research with respect to our autonomous spacecraft architecture. First, our architecture could benefit from an increased use of simulation. Currently we use a simulator for development and testing the software. This could be extended to facilitate interactive knowledge acquisition and refinement, to improve projection in the planner, or to provide a tighter integration between planning and execution (Drummond, Bresina, & Swanson 1994; Levinson 1994). Second, our architecture leaves open issues of machine learning, which could be used to tune parameters in the control system, for optimizing search control in planning, or for modifying method selection priorities during execution. Third, we see substantial benefits in having a single representation of the spacecraft, supporting multiple uses by processes of abstraction and translation. We believe that progress toward this goal is best made by generalizing from powerful, focused models capable of representing the complexities of a real-world domain.

References

[1] Ahmed, A.; Aljabri, A. S.; and Eldred, D. 1994. Demonstration of on-board maneuver planning using autonomous s/w architecture. In *8th Annual AIAA/USU Conference on Small Satellites*.

[2] Brown, G.; Bernard, D.; and Rasmussen, R. 1995. Attitude and articulation control for the cassini spacecraft: A fault tolerance overview. In *14th AIAA/IEEE Digital Avionics Systems Conference.*

[3] de Kleer, J., and Williams, B. C. 1987. Diagnosing multiple faults. *Artificial Intelligence* 32(1):97–130. Reprinted in (Hamscher, Console, & de Kleer 1992).

[4] de Kleer, J., and Williams, B. C. 1989. Diagnosis with behavioral modes. In *Proceedings of IJCAI-89*, 1324–1330. Reprinted in (Hamscher, Console, & de Kleer 1992).

[5] de Kleer, J., and Williams, B. C., eds. 1991. *Artificial Intelligence*, volume 51. Elsevier.

[6] Drummond, M.; Bresina, J.; and Swanson, K. 1994. Just-in-case scheduling. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, 1098–1104. Cambridge, Mass.: AAAI.

[7] Firby, R. J. 1978. *Adaptive execution in complex dynamic worlds.* Ph.D. Dissertation, Yale University.

[8] Gat, E. 1992. Integrating planning and reacting in a heterogeneous asynchronous architecture for controlling real-world mobile robots. In *Proceedings of the Tenth National Conference on Artificial Intelligence.* Cambridge, Mass.: AAAI.

[9] Gat, E. 1996. Csl: A language for supporting robust plan execution in autonomous spacecraft. In preparation.

[10] Georgeff, M. P., and Lansky, A. L. 1987. Procedural knowledge. Technical Report Technical Note 411, Artificial Intelligence Center, SRI International.

[11] Hackney, J.; Bernard, D.; and Rasmussen, R. 1993. The cassini spacecraft: Object oriented flight control software. In *1993 Guidance and Control Conference.*

[12] Hamscher, W.; Console, L.; and de Kleer, J. 1992. *Readings in Model-Based Diagnosis.* San Mateo, CA: Morgan Kaufmann.

[13] Hayes-Roth, B. 1995. An architecture for adaptive intelligent systems. *Artificial Intelligence* 72.

[14] Laird, J. E.; Newell, A.; and Rosenbloom, P. S. 1987. Soar: An architecture for general intelligence. *Artificial Intelligence* 33(1).

[15] Levinson, R. 1994. A general programming language for unified planning and control. *Artificial Intelligence.* Special Issue on Planning and Scheduling.

[16] Muscettola, N. 1994. HSTS: Integrating planning and scheduling. In Fox, M., and Zweben, M., eds., *Intelligent Scheduling.* Morgan Kaufmann.

[17] Musliner, D.; Durfee, E.; and Shin, K. 1993. Circa: A cooperative, intelligent, real-time control architecture. *IEEE Transactions on Systems, Man, and Cybernetics* 23(6).

[18] Noreils, F., and Chatila, R. 1995. Plan execution monitoring and control architecture for mobile robots. *IEEE Transactions on Robotics and Automation.*

[19] R. P. Bonasso, D. Kortenkamp, D. M., and Slack, M. 1995. Experiences with an architecture for intelligent, reactive agents. In *IJCAI Workshop on Agent Theories, Architectures, and Languages*. IJCAI.

[20] R. P. Bonasso, D. Kortenkamp, D. M., and Slack, M. 1996. Experiences with an architecture for intelligent, reactive agents. *JETAI.* to appear.

[21] Simmons, R. 1990. An architecture for coordinating planning, sensing, and action. In *Proceedings DARPA Workshop on Innovative Approaches to Planning, Scheduling and Control*, 292–297. Morgan Kaufmann: San Mateo, CA.

[22] Tambe, M.; Johnson, W. L.; Jones, R. M.; Koss, F.; Laird, J. E.; Rosenbloom, P. S.; and Schwamb, K. 1995. Intelligent agents for interactive simulation environments. *AI Magazine* 16(1):15–39.

[23] Weld, D. S., and de Kleer, J., eds. 1990. *Readings in Qualitative Reasoning About Physical Systems*. San Mateo, California: Morgan Kaufmann Publishers, Inc.

[24] Wilkins, D. E., and Myers, K. L. 1995. A common knowledge representation for plan generation and reactive execution. *Journal of Logic and Computation.*