Proceeding of the 6th International Symposium on Artificial Intelligence and Robotics & Automation in Space: i-SAIRAS 2001, Canadian Space Agency, St-Hubert, Quebec, Canada, June 18-22, 2001.

A Reactive Model-based Programming Language for Robotic Space Explorers

Michel Ingham, Robert Ragno, Brian Williams
MIT Space Systems Laboratory / Artificial Intelligence Laboratory
Cambridge, MA 02139
{ingham, rjr, williams}@mit.edu

Keywords model-based autonomy, model-based programming, execution, synchronous programming, automated reasoning

Abstract

Model-based autonomous agents have emerged recently as vital technologies in the development of highly autonomous reactive systems, particularly in the aerospace domain. These agents utilize many automated reasoning capabilities, but are complicated to use because of the variety of languages employed for each capability. To address this problem, we introduce *model-based programming*, a novel approach to designing embedded software systems. In particular, we introduce the Reactive Model-based Programming Language (RMPL), which provides a framework for constraint-based modeling, as well as a suite of reactive programming constructs. To convey the expressiveness of RMPL, we show how it captures the main features of synchronous programming languages and advanced robotic execution languages. This paper focuses on using the rich behavior modeling of RMPL to provide sequencing and robotic execution capabilities for spacecraft.

1 Introduction

Several highly autonomous aerospace systems have been recently deployed, and more are currently under development. Examples include NASA's Deep Space One (DS1) spacecraft [1], the next generation of space telescopes [2], and Mars rover prototypes [3]. These systems integrate many of the tools of AI research for automated reasoning: planning and scheduling, task decomposition execution, model-based reasoning and constraint satisfaction.

Though these technologies show great promise, there is a likely barrier to the widespread deployment of this level of autonomy: the numerous software tasks running on a spacecraft processor are generally encoded using a variety of modeling and programming languages. These tasks include sequencing, system monitoring, system reconfiguration, planning and low-level control. To illustrate this problem, consider the current state-of-the-art in model-based autonomy software, Remote Agent (RA). Flight-validated on the DS1 spacecraft in 1999, RA is composed of three primary components: a Planner/Scheduler, a Smart Executive, and a Mode Identification and Reconfiguration module known as Livingstone.

The complexity of software interfaces was identified as a key challenge encountered during the integration of the RA and the flight software [4]. Furthermore, each of the three component technologies requires a distinct knowledge representation, expressed using different modeling languages. While such heterogeneous representations have a number of benefits, including the ability for different software components to reason at different levels of abstraction, they also raise several difficulties. Most significant are the possibility for models to diverge, and the need to duplicate knowledge representation efforts. One conclusion from the RA design effort is the desire to head towards a unified representation of the spacecraft, while maintaining the ability to accommodate the complexities of the spacecraft domain, and the capacity for knowledge abstraction [5].

To address this desire, we introduce the notion of *model-based programming*, a novel approach to designing embedded software systems. We also introduce a language for encoding model-based programs, the *Reactive Model-based Programming Language* (RMPL). RMPL combines constraint-based modeling with reactive programming constructs, and offers a simple model of computation in terms of hierarchical constraint-based automata. Systems that monitor, diagnose and plan using RMPL are discussed in [6] and [7]. The goal of this paper is to show how RMPL leverages the features of both embedded synchronous programming languages and advanced robotic execution systems, to provide a framework for robust spacecraft sequencing. We also show how RMPL forms the basis for a model-based executive providing integrated monitoring, fault protection and sequencing capabilities.

In Section 2, we first introduce the model-based programming paradigm. The RMPL language is then introduced in Section 3, by describing its principal constructs and its underlying model of computation. In Section 4, we discuss the compilation and execution of model-based programs, and show how a model-based executive can be used to enable robust spacecraft autonomy. To convey the expressiveness of RMPL, in Section 5 we show how it captures the main features of synchronous programming languages and advanced robotic execution languages. Finally, we conclude by highlighting future work that will allow RMPL to be applied to other tasks in automated reasoning.

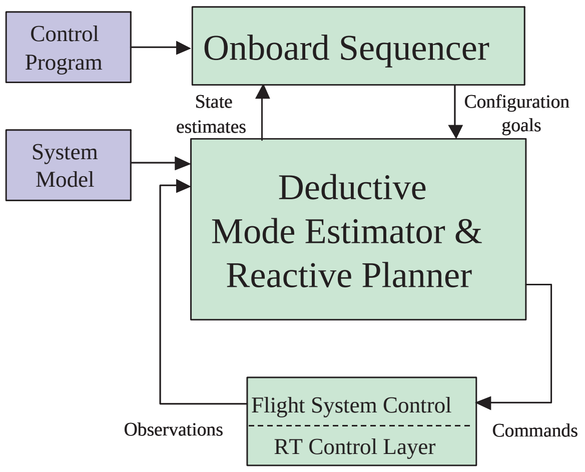


Figure 1: Architecture for a Model-based Executive

2 Model-based Programming

In this section we introduce the notion of model-based programming as an approach to writing software for embedded reactive systems. Figure 1 illustrates the model-based programming paradigm, as embodied in a model-based executive. The underlying principle is that control programs 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. Such a control program is input to a sequencing engine, for onboard execution. An underlying mode estimation and reactive planning layer uses a model of the system to deduce the system state from the observables, and to figure out how to achieve a specified goal state. This deductive layer can reason about states of physical components in the system, or abstract states representing subsystem-level behavior.

Model-based programming shares some common architectural themes with the Mission Data System (MDS), under development at the Jet Propulsion Laboratory [8]. In particular, both concepts advocate that *system state* and *models* form the foundation for monitoring and control. Model-based programming concentrates on providing a programming language centered on hidden state, and seamlessly incorporating powerful model-based deductive engines into the program interpreter.

Model-based programming offers the following advantages over traditional approaches to embedded software development:

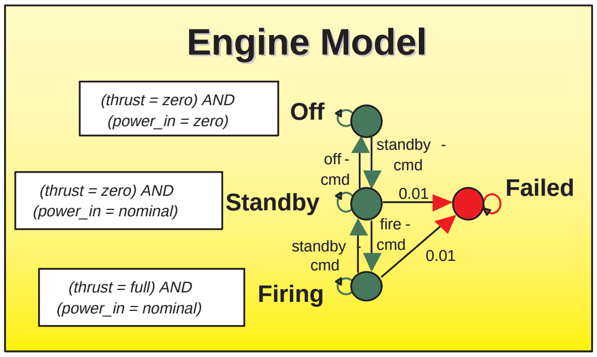
Abstraction - By abstracting away the details of how a particular state is inferred or achieved, the system engineer’s knowledge of spacecraft behavior (e.g. as represented in StateCharts [9]) can be converted into flight code in a more straightforward manner. This is because it is easier to specify a desired state than the sequence of control actions needed to reach it.

Powerful Inference Engines - The model-based programming paradigm leaves the challenging tasks of state estimation and state achievement in the hands of powerful model-based deduction engines, such as those introduced in [10] and [11]. Such engines offer more flexibility and robustness than the traditional rule-based engines commonly deployed in flight software.

Modularity - Because of its use of modular system models, flight software written as a model-based program can accommodate component-level modifications late in the spacecraft design cycle. New component models can be swapped in without having to rewrite significant sections of flight code. Furthermore, modularity within the deductive layer allows for transparent upgrading of the engines used for mode estimation and reactive planning, when more powerful ones become available.

Model Reusability - Another benefit to the use of modular system models is that the component-level models can be reused. Over time, a database of models for different subsystems and component designs can be assembled, dramatically reducing the need for single-use flight code.

Verifiability - As a consequence of being able to write control code in terms of states, model-based programming results in cleaner code that is easier to verify. Ease of verification is also a feature of the system models, which can be built up directly from system engineering specifications of hardware or software components.



Engine Model
(thrust = zero) AND (power_in = zero)
Off
(thrust = zero) AND (power_in = nominal)
Standby
(thrust = full) AND (power_in = nominal)
Firing
Failed
standby - cmd
off - cmd
0.01
fire - cmd
standby - cmd
0.01

Figure 2: State transition model for an engine

```
OrbitInsert ()::
(do-watching ((EngineA = Firing) OR (EngineB = Firing))
    (parallel
        (EngineA = Standby)
        (EngineB = Standby)
        (Camera = Off)
        (do-watching (EngineA = Failed)
            (when-donext ( (EngineA = Standby) AND
                           (Camera = Off) )
                (EngineA = Firing) ))
        (when-donext ( (EngineA = Failed) AND
                       (EngineB = Standby) AND
                       (Camera = Off) )
            (EngineB = Firing) )))
```

Figure 3: RMPL Control Program for the Orbital Insertion Scenario

Model-based Programming Example

To demonstrate the applicability of model-based programming to the field of spacecraft execution, we consider the representative mission scenario of performing an orbital insertion maneuver. As shown in Figure 1, a model-based program specifies two inputs, a control program and a system model. The system model is composed of component-level models. In our example, we consider a simplified spacecraft model, consisting of two identical redundant engines (*EngineA* and *EngineB*) and a science *Camera*. The state transition model for an engine is shown in Figure 2. Nominally, an engine can be in one of three modes: *off*, *standby*, or *firing*. Each of these modes is described by a set of constraints on the *thrust* and *power_in* variables. The engine also has a *failed* mode, capturing any off-nominal behavior. Transitions between nominal modes are triggered on commands, as shown in the figure. Transitions into the *failed* mode are considered to be probabilistic, with a 1% chance of failure from both the *standby* and *firing* modes. The *Camera* can be in one of two modes, *on* or *off*, with corresponding "turn-on" and "turn-off" transitions. This type of transition system model is similar to the one used in Livingstone [10].

The control program in Figure 3 specifies state trajectories for the orbital insertion maneuver. The specific RMPL constructs used in the program are introduced in Section 3. In order to execute the orbital insertion, both engines must be set to standby mode and one of the two engines must be fired. Prior to firing an engine, the camera must be turned off, for fear of plume contamination of its sensitive optics. Should the first engine fail to fire correctly, the backup engine should be commanded to fire instead. To express this, our control program must be able to represent the following types of behavior:

1. **conditional branching** - The control code must check for two conditions, prior to firing its engine: that the engine to be fired is in standby mode, and that the camera is turned off.
2. **iteration** - Should the above two conditions take some time to be achieved, the control program should wait for both to become true, requiring an iteration on the condition check. Also, while the primary engine is firing, there must be a continuous check for engine failure.
3. **preemption** - If the primary engine fails, the act of firing it should be preempted, in favor of firing the backup engine.
4. **concurrency** - The following tasks should be performed in parallel: setting each engine to standby mode, turning off the science camera, checking for completion of these actions prior to firing the primary engine, and checking for primary engine failure.

These types of behaviors are common to embedded programming. The key distinction in our model-based control programs is that we refer directly to hidden variables in the plant. Furthermore, we would like to express the spacecraft state transition models described above using a language common to both the system models and the control program. Such a language would need to be able to specify deterministic and probabilistic transitions, as well as qualitative constraints describing the behavior in each mode. In the next section, we introduce a language expressive enough to encode all these types of behavior.

3 The RMPL Language

RMPL is an object-oriented language that allows a domain to be structured in a variety of ways – for example, through a component or process hierarchy. RMPL enables modeling of complex elements of a domain through an object hierarchy that includes subclassing and multiple inheritance.

RMPL programs may be viewed as specifications of deterministic state transition systems, which act on the plant by asserting and checking constraints expressed in a propositional state logic. The propositions are assignments of state variables to values within their domains. Reactive combinators allow flexibility in expression of complex system behavior and dynamic relations.

3.1 RMPL Constructs

The constructs of RMPL are similar to those developed in TCC, a language for timed concurrent constraint programming [12]. RMPL constructs are stated as operator-prefixed, parenthesis-enclosed expressions. These expressions may be arbitrarily composed with each other. An RMPL program is a single expression built up from simpler expressions. The constructs rely on the system state knowledge, consisting of a set of value assignments to all state variables. A constraint is *entailed* if it is implied by the current state knowledge, and *not entailed* otherwise. Entailment is denoted by simply stating the constraint, non-entailment is denoted by using an overbar. Note that non-entailment of c ($\bar{c}$) is *not* equivalent to entailment of the negation of c (NOT c) – the current state knowledge may not imply c to be true or false. The following are the key constructs in the language:

c*onstraint*
Constraint is expressed as a well-formed-formula in state logic. The simplest constraint expression is an assertion of the state of a variable, such as ($thrust = full$). More complex constraints can be created using the logical connectives NOT, AND, and OR. The constants $true$ and $false$ are also valid constraints. Note that the same form is used for both checking constraints and asserting constraints.

(PARALLEL exp_1 exp_2 **...)**
Concurrency. Execution of each expression exp_i is initiated concurrently.

(SEQUENCE exp_1 exp_2 **...)**
Sequence. Starting with $i = 0$, executes expression exp_i until completion, and then initiates expression exp_{i+1} in the next time step. Terminates when the last expression terminates.

(ALWAYS *exp***)**
Iteration. Expression *exp* is initiated at every time step.

(IF-THENNEXT-ELSENEXT c exp_{then} exp_{else}**)**
Branch. If constraint c is entailed, then executes expression exp_{then} in the next time step. Otherwise (if c is not entailed), executes expression exp_{else} in the next time step.

(UNLESS-THENNEXT *constraint exp***)**
Guarded transition. Unless *constraint* is entailed, expression *exp* is executed in the next time step.

(WHEN-DONEXT *constraint exp***)**
Temporally-extended guarded transition. Wait until *constraint* is entailed, then execute expression *exp* in the next time step.

(WHENEVER-DONEXT *constraint exp***)**
Iterated guarded transition. Wait until *constraint* is entailed, then execute expression *exp* in the next time step. This trigger can repeat indefinitely.

(DO-WATCHING *constraint exp***)**
Preemption. Executes expression *exp* until completion or until *constraint* is entailed. If and when *constraint* is entailed, the execution of expression *exp* is halted at the beginning of the step in which *constraint* is entailed.

Constructs also exist to represent probabilistic and reward-producing transitions [6]. Other higher-level constructs can be constructed from the above.

3.2 Hierarchical Constraint Automata

To support efficient execution or reasoning, RMPL code is compiled into hierarchical constraint automata (HCA). Figure 4 shows the HCA structures represented by various RMPL constructs. HCA consist of nodes (called *locations*) that may assert constraints, and directed edges (called *transitions*) that may have associated guard conditions. Groups of locations may also have associated maintenance constraints. The transition guards and maintenance constraints may either indicate entailment or non-entailment. HCA are hierarchical in that a transition may connect a location to either another location or an HCA. A subset of the locations in an HCA are designated as start locations. These start locations serve as "entry points" into the automaton during their execution. The HCA execution process is described in the following section.

4 Compilation and Execution of Model-based Programs

In this section, we show how a control program written in RMPL is compiled to HCA, and discuss how an HCA is executed. To illustrate this process clearly, we return to our motivating example from Section 2. Figure 3 shows how the orbital insertion control code looks when encoded in RMPL. The RMPL compiler takes this code as an input, and outputs the corresponding HCA system (see Figure 5). It should be noted that the compilation process takes place offline. Only the resulting HCA model is loaded onto the onboard flight processor for eventual execution.

The current execution threads in the HCA system are represented by a set of "marked" locations. Execution of an HCA consists of stepping through the

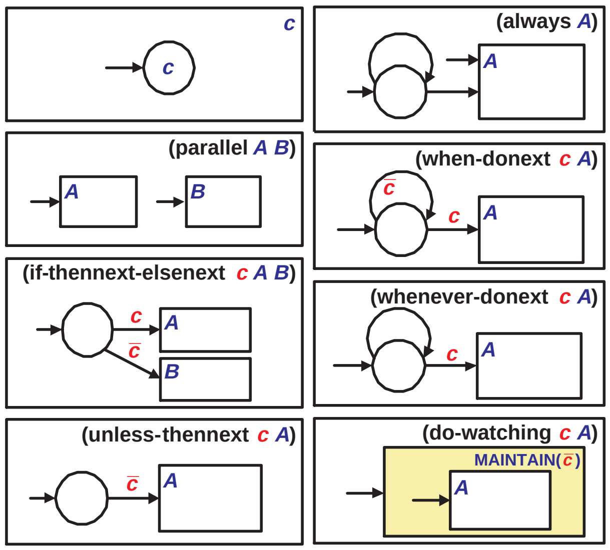


Figure 4: HCA representations of various RMPL forms.

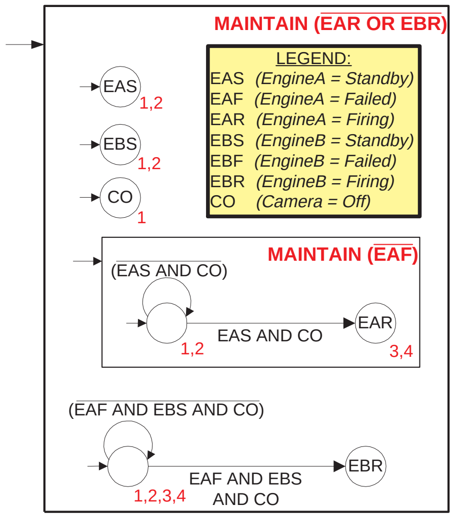


Figure 5: HCA model for the Orbital Insertion Scenario (markings for nominal execution are labeled in red)

HCA, beginning from the start locations, and taking all enabled transitions from each currently marked location. Steps are performed synchronously, with all possible transitions assumed to be simultaneous. If a location transitions to an HCA, that HCA is initialized (by marking its start locations). This allows the initiation of components of the hierarchical structure.

The algorithm for executing an HCA model can be described as follows:

1. Initialize HCA system by marking all start locations.
2. Assert states from the currently marked locations (as well as any active maintenance conditions specified by an encapsulating "do-watching"). The reactive planner deduces a sequence of actions to *eventually* achieve each of the asserted system states, and issues the first set of commands in the sequence.
3. Obtain from the mode estimation module an update of the system states.
4. Take enabled transitions. A location's transitions are enabled if (1) the location's state assignment has been achieved, and (2) the transition conditions and maintenance conditions currently hold true.
5. Mark new set of locations, delete previous set of expired markings, and return to step 2.

HCA Execution Example

To illustrate the process of executing HCA, we consider a nominal (i.e. failure-free) execution trace from our example scenario. In Figure 5, markings are represented by small numbered labels adjacent to each location. The numbers correspond to the order of execution steps, e.g. the location which asserts the state ($EngineA$ = $Standby$) is marked in both the first and second steps.

Initially, all start locations are marked. The start locations are those labeled with "1". The following state assignments are asserted by the start locations: *EngineA=Standby*, *EngineB=Standby* and *Camera=Off*. The maintenance condition to be checked throughout the process of achieving these states is the non-entailment of *(EngineA=Firing OR EngineB=Firing)*. These state assignments and maintenance conditions are sent to the reactive planner, which generates the sequence that achieves the goal state, and issues the first set of commands in the sequence.

The enabled transitions of the currently marked locations are then taken. In our example, we assume that turning a camera off is a one-step operation, whereas putting an engine into standby mode requires two steps (the intermediate step might be to turn on a valve driver, for instance). Consequently, the locations asserting *(EngineA=Standby)* and *(EngineB=Standby)* remain marked in the next execution step (labeled "2" in Figure 5). Since *Camera=Off* has been achieved within a single step (as confirmed by the mode estimator), and there are no specified transitions from the location that asserted this state, this thread of execution terminates. The last two markings from execution step "1", corresponding to "when-donext" condition checks, remain marked at step "2", as the only currently enabled transitions from these locations are the selftransitions.

At step "2", the reactive planner issues the second set of actions in the sequence for achieving *(EngineA=Standby)* and *(EngineB=Standby)*. Assuming these actions are successfully performed, the mode estimation module indicates that the *(EngineA=Standby)* and *(EngineB=Standby)* conditions are achieved. This results in termination of the two execution threads corresponding to these state achievement requests, and enabling of the transition labeled with the condition *(EngineA=Standby AND Camera=Off)*. Note that the maintenance conditions associated with this transition, corresponding to non-entailment of *(EngineA=Failed)*, and non-entailment of *(EngineA=Firing OR EngineB=Firing)*, still hold true.

After taking the enabled transitions, only two locations are marked at step "3". One of these marked locations asserts *(EngineA=Firing)*. This state assignment is sent to the reactive planner, which generates the two-step sequence that achieves this state, and issues the first set of commands in the sequence.

Finally, at step "4" the reactive planner issues the second set of commands required to achieve *(EngineA=Firing)*, and the mode estimation module infers that the engine is indeed firing. Since this violates the high-level maintenance condition being watched for, the entire block of Figure 5 is exited. Note that since no engine failure occurred during this process, the thread of execution waiting for *(EngineA=Failed)* did not advance from its start location.

5 Expressiveness of RMPL

To serve as the foundation for the development of a model-based execution framework, RMPL must provide certain key features of other languages used for embedded robotic systems. Expressiveness is achieved through the combination of constraint-based modeling and recent advances in two different domains: synchronous programming, used in many industrial embedded reactive systems, and robotic execution languages, used to provide robust sequencing capabilities for ground-based robots and autonomous spacecraft.

In this section, we highlight the relevant aspects of each of these two domains to the design of RMPL, and provide a rough mapping of these features to constructs in RMPL.

5.1 Synchronous Programming Languages

The field of synchronous programming offers a class of languages developed for writing control programs for embedded reactive systems. Esterel, Lustre and Signal are synchronous programming languages that have been deployed for industrial applications [13, 14]. The widely used StateCharts graphical specification formalism [9] shares key aspects of the synchronous programming model. In the design of RMPL, certain key ideas from the synchronous programming domain have been leveraged. In this section, we highlight the similarities and fundamental differences between RMPL, and one widely-used synchronous language, Esterel.

First we consider similarities between the two languages. Both include standard constructs for expressing reactive system behavior, such as conditional branching, iteration, parallel composition, sequential ordering and preemption. Figure 6 provides a mapping between constructs in Esterel and the corresponding RMPL forms. Both languages compile to underlying automaton models with clean mathematical semantics. They both emphasize modularity in software design: Esterel uses the 'module' as its programming unit, while RMPL uses the notion of 'class' objects. Finally, like Esterel, RMPL is fully orthogonal, meaning that the constructs can be nested and combined arbitrarily.

Despite the similarities between the two languages, there exist some fundamental differences in their corresponding philosophy:

- Esterel is very much a "signal-based" language, whereas RMPL is a "state-based" language. Rather than thinking in terms of *signals*, as is the case for Esterel, we consider the notion of hidden system *states* to be the fundamental basis for execution with RMPL. This different point of view is attributable to the different application domains targeted by the two languages. For Esterel, which is designed to provide coordination and synchronization between computational processes, signals and events are the most appropriate basic mechanisms. RMPL's task, which is to provide a framework for monitoring and control of complex dynamic plants, lends itself to thinking more naturally in terms of state evolution.
- The availability of instantaneous broadcasting and control transmission in Esterel makes it possible to write syntactically correct but semantically "non-sensical" programs [13]. RMPL's more strict use of execution steps avoids these causality problems.
- The intent of Esterel is to provide the functionality of a full programming language, including external function and procedure calls, and external task execution. RMPL's scope is focused towards providing the functionality required to evolve the state of a complex system.

Construct Description	Esterel	RMPL
1-step delay	`pause`	`true`
halting	`halt`	`(ALWAYS true)`
sequencing	`body1 ; body2`	`(SEQUENCE body1 body2)`
conditional	`if exp then body1 else body2 end`	`(IF-THENNEXT-ELSENEXT exp body1 body2)`
concurrency	`body1 || body2`	`(PARALLEL body1 body2)`
await	`await S`	`(DO-WATCHING S (ALWAYS true))`
await-do	`await S do body end`	`(WHEN-DONEXT S body)`
trap definition	`trap id in body end`	`(DO-WATCHING (exit=true) body)`
weak abort	`weak abort body when S`	`(DO-WATCHING (exit=true) (PARALLEL body (WHEN-DONEXT S (exit=true))))`
abort with handler	`abort body1 when S do body2 end`	`(PARALLEL (DO-WATCHING S body1) (WHEN-DONEXT S body2))`
guarded loop	`every S do body end`	`(WHENEVER-DONEXT S (DO-WATCHING S body))`

Figure 6: Corresponding constructs in Esterel and RMPL

It currently relies less heavily on code written in an external "host" language. Future deployment of RMPL to various applications will reveal whether the additional capability provided by such external links warrants extension of the language.

5.2 Robotic Execution Languages

State-of-the-art spacecraft sequencers, like the Smart Executive flown as part of the Remote Agent Experiment on Deep Space One [15], are used to coordinate the run-time activity among the different software modules within a control system. They must be able to respond quickly to events while bringing potentially large amounts of information (both knowledge and run-time data) to bear on their decisions [16]. Robust executives are becoming even more essential as autonomous agents of increasing capability are developed. The robotic execution languages used to encode advanced sequencing engines (e.g. RAPs [17], TDL [18]) provide constructs for managing interacting parallel goal- and event-driven processes.

We consider the example of the Remote Agent Executive. It is written in a rich procedural language, Execution Support Language (ESL), which is an extension to multi-threaded Common Lisp. ESL provides the following features, allowing the encoding of execution knowledge into embedded autonomous agents [16]: contingency handling, timekeeping, task management, goal achievement, logical database querying, and resource management.

Through its rich set of reactive constructs, RMPL provides most of these features. By combining its preemption constructs (e.g. "do-watching") with the hidden state diagnosis capability provided by the underlying deductive layer, mechanisms for identifying and recovering from failure contingencies and specifying cleanup procedures can be built.

Task management capabilities, such as spawning new concurrent tasks, aborting tasks, setting up task networks and defining guardian tasks (for task monitoring) are provided by RMPL's parallel composition and preemption constructs. Task synchronization features, including signaling and waiting for particular events, take on a different meaning in a model-based programming language like RMPL: as long as the "events" can be represented as changes to system states, they are straightforwardly accommodated.

The mechanisms for specifying and commanding goal achievement methods are provided by the underlying reactive planning layer, in the model-based programming framework. The capacity for a model-based executive to perform sequencing at the level of system state specifications, and to abstract away the details of how states are achieved, is considered a significant benefit, as discussed in Section 2.

Similarly, the need to explicitly maintain, query and reason about a distinct logical database is made obsolete by the presence of a transparent underlying mode estimation layer, which maintains the latest system state knowledge, and provides the sequencer with all the state information it requires.

The dominant features that the current incarnation of RMPL cannot adequately provide are time-related utilities (such as event scheduling, timeout definitions or warping), and advanced property locking mechanisms (resource interactions between concurrent tasks). It should be noted that RMPL does provide the ability to perform basic resource management, using the preemptive constructs to trigger when a certain property, expressed in terms of state assignments, is no longer entailed. Incorporation of timing-related constructs and more advanced resource management features is a focus of current work.

6 Conclusion

We have discussed the design of a model-based executive, consisting of a sequencing layer coded in RMPL and compiled down to HCA, and an underlying deductive layer providing mode estimation and reactive planning capabilities, based on models of the system expressed in RMPL. In the current design of the model-based executive, the sequencing layer and the deductive layer are distinct. One of the eventual goals of this research task is the integration of both capabilities into a unified system, which will allow us to perform monitoring, diagnosis and reconfiguration on the same HCA models that are executed, eliminating the need for separately maintaining the control program and the system models.

The ultimate goal of this research effort is to arrive at a fully integrated autonomous system, which uses a unified system representation to provide high-level planning and scheduling capabilities, in addition to robust sequencing and configuration management. To this end, a temporal planning capability based on an extension of RMPL (including the notions of metric time, temporally bounded activity specifications and non-deterministic choice) is currently under development [7]. In addition, to address the problem of monitoring and controlling systems which must be described using continuous dynamics, a hybrid version of RMPL is under development [19]. This Hybrid Model-based Programming Language incorporates the notion of continuous variables with RMPL's reactive programming and constraint-based modeling features.

7 Acknowledgments

This research was supported in part by the DARPA MOBIES program under contract F33615-00-C-1702, and by NASA's Cross Enterprise Technology Development program under contract NAG2-1466.

References

[1] Muscettola, N. et al., "Remote Agent: To Boldly Go Where No AI System Has Gone Before", *Artificial Intelligence*, 103(1-2):5-48, 1998.

[2] Ingham, M. et al., "Autonomous Sequencing and Model-based Fault Protection for Space Interferometry", *Proc. of iSAIRAS-2001*, 2001.

[3] Washington, R. et al., "Autonomous Rovers for Mars Exploration", *Proc. of the IEEE Aerospace Conference*, 1999.

[4] Nayak, P.P. et al., "Validating the DS1 Remote Agent Experiment", *Proc. of iSAIRAS-1999*, 1999.

[5] Bernard, D.E. et al., "Design of the Remote Agent Experiment for Spacecraft Autonomy", *Proc. of the IEEE Aerospace Conference*, 1999.

[6] Williams, B.C., Chung, S., and Gupta, V., "Mode Estimation of Model-based Programs: Monitoring Systems with Complex Behavior", To appear in *Proc. of IJCAI-2001*, 2001.

[7] Kim, P., Williams, B.C., and Abramson, M., "Executing Reactive, Model-based Programs through Graph-based Temporal Planning", To appear in *Proc. of IJCAI-2001*, 2001.

[8] Dvorak, D. et al., "Software Architecture Themes in JPL's Mission Data System", *Proc. of the AIAA Guidance, Navigation, and Control Conference and Exhibit*, Portland, OR, 1999.

[9] Harel, D., "Statecharts: A Visual Approach to Complex Systems", *Science of Computer Programming* 8:231-274, 1987.

[10] Williams, B.C. and Nayak, P.P., "A Model-based Approach to Reactive Self-Configuring Systems", *Proc. of AAAI-96*, pp.971-978, 1996.

[11] Williams, B.C. and Nayak, P.P., "A Reactive Planner for a Model-Based Executive", *Proc. of IJCAI-97*, 1997.

[12] Saraswat, V., Jagadeesan, R., and Gupta, V., "Foundations of Timed Concurrent Constraint Programming", *Proc. of the Ninth Annual IEEE Symposium on Logic in Computer Science*, 1994.

[13] Berry, G., *The Esterel v5 Language Primer*, version 5.21 release 2.0, Centre de Mathematiques Appliquees, Ecole des Mines and INRIA, April 6, 1999.

[14] Halbwachs, N., *Synchronous Programming of Reactive Systems*, Kluwer Academic Publishers, 1993.

[15] Pell, B. et al., "The Remote Agent Executive: Capabilities to Support Integrated Robotic Agents", *Proc. of the AAAI Spring Symposium on Integrated Robotic Architectures*, 1998.

[16] Gat, E., "ESL: A Language for Supporting Robust Plan Execution in Embedded Autonomous Agents", *Proc. of the AAAI Fall Symposium on Plan Execution*, 1996.

[17] Firby, R.J., *Adaptive Execution in Dynamic Domains*, Ph.D. Thesis, Yale University Department of Computer Science, 1989.

[18] Simmons, R. and Apfelbaum, D., "A task Description Language for Robot Control", *Proc. of the Conference on Intelligent Robotics and Systems*, Vancouver, Canada, October 1998.

[19] Williams, B., Hofbaur, M., and Jones, T., *Mode Estimation of Probabilistic Hybrid Systems*, MIT Space Systems Laboratory Report #6-01, Massachusetts Institute of Technology, May 2001.