Unifying Model-based and Reactive Programming within a Model-based Executive

Brian C. Williams and Vineet Gupta [†]
NASA Ames Research Center, MS 269-2
Moffett Field, CA 94035 USA
E-mail: {williams, vgupta}@ptolemy.arc.nasa.gov

Abstract

Real-time model-based deduction has recently emerged as a vital component in AI's tool box for developing highly autonomous reactive systems. Yet one of the current hurdles towards developing model-based reactive systems is the number of methods simultaneously employed, and their corresponding melange of programming and modeling languages. This paper offers an important step towards unification of reactive and model-based programming, providing the capability to monitor mixed hardware/software systems. We introduce *RMPL*, a rich modeling language that combines probabilistic, constraint-based modeling with reactive programming constructs, while offering a simple semantics in terms of hidden state Markov processes. We introduce *probabilistic, hierarchical constraint automata*, which allow Markov processes to be expressed in a compact representation that preserves the modularity of RMPL programs. Finally, a model-based executive, called RBurton is described that exploits this compact encoding to perform efficent simulation, belief state update and control sequence generation.

Introduction

Highly autonomous systems, such as NASA's Deep Space One spacecraft(Muscettola *et al.* 1999) and Rover prototypes, leverage many of the fruits of AI's work on automated reasoning – planning and scheduling, task decomposition execution, model-based reasoning and constraint satisfaction. Yet a likely show stopper to widely deploying this level of autonomy is the multiplicity of AI modeling languages employed for tasks such as diagnosis, execution, control and planning for these systems. Currently separate models of the same artifacts are built for each of these tasks, leading to considerable overhead on the part of the modeling teams — for the Deep Space One spacecraft, as much as half the total development time was devoted to ensuring that the multiple models were mutually consistent.

This paper concentrates on a developing a unified language for monitoring, diagnosis and recovery, and reactive execution. Key to this challenge is the development of a unified language that can express a rich set of mixed hardware *and* software behaviors (Reactive Model-based Programming Language - RMPL), a compact encoding of the underlying Markov process (hierarchical constraint automata – HCA), and an executive for this encoding that supports efficient state estimation, monitoring and control generation (RBurton).

RMPL. RMPL achieves expressiveness by merging key ideas from synchronous programming languages, qualitative modeling and Markov processes. Synchronous programming offers a class of languages (Halbwachs 1993) developed for writing control programs for reactive systems (Harel & Pnueli 1985; Berry 1989). Qualitative modeling and Markov processes provide means for describing continuous processes and uncertainty.

RBurton achieves efficient execution through a careful generalization of state enumeration algorithms that are successfully employed by the Sherlock(de Kleer & Williams 1989) and Livingstone(Williams & Nayak 1996) systems on simpler modeling languages.

Monitoring software/hardware systems. The applications for which modeling in RMPL will be necessary are mixed hardware software systems. In any complex system, the overall control system and its individual components are mixed hardware software systems. With respect to diagnosis this means that tracking the system's behavior requires tracking the software execution at some level of abstraction. In addition, failures can effect the execution trace of the software and hence symptoms might manifest themselves as software error conditions.

A key feature of RMPL is that the execution and control software can also be written in RMPL, thus we can do our monitoring and diagnosis on the same models that are executed — "What you monitor is what you execute", a variant of Berry's principle*. This eliminates the need for separately maintaining the execution

[†] Caelum Research Corporation.

* "What you prove is what you execute"(Berry 1989). Berry argues that verification must be done directly on executable programs, eliminating the gap between specifications about which we prove properties, and the programs that are supposed to implement them.

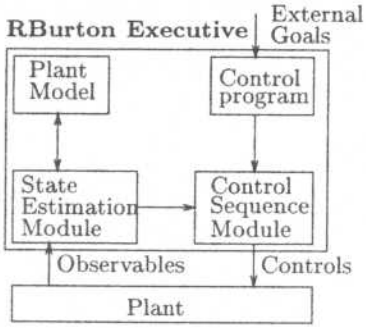
software and models of it for diagnosis, a potentially expensive and error-prone task.

Organization of this paper. We start with a sketch of RBurton. The first half of the paper then introduces hierarchical constraint automata, their deterministic execution, and their expression using RMPL. The direct mapping from RMPL combinators to HCA, coupled with HCA's hierarchical representation avoids the state explosion problem that frequently occurs while compiling large reactive programs.

The second half of the paper turns to model-based execution under uncertainty. First we generalize HCAs to a factored representation of partially observable Markov processes. We then develop RBurton's stochastic monitoring and execution capabilities, while leveraging off the compact encoding offered by probabilistic HCA. The paper concludes with a discussion of related work.

The RBurton Executive

The execution task consists of controlling a physical plant according to a stream of high-level commands (goals), in the face of unexpected behavior from the system. To accomplish this the executive controls some variables of the plant, and senses the values of some sensors to determine the state of the plant.



A schematic of RBurton is shown above. RBurton consists of two main components. The state estimation module determines the current most likely states of the plant from observed behavior using a plant model. This generalizes mode identification (MI), (Williams & Nayak 1996). The key difference is the expressiveness of the modeling languages employed. RMPL allows a rich set of embedded software behaviors to be modeled, hence RBurton's state estimator offers a powerful tool for monitoring mixed software/hardware systems.

RBurton's control sequencer executes a program for controlling the plant that is also specified using RMPL. Actions are conditioned on external goals and properties of the plant's current most likely state. Given multiple feasible options, RBurton selects the course of action that maximizes immediate reward. As a control language RMPL offers the expressiveness of reactive languages like Esterel(Berry & Gonthier 1992), along with many of the goal-directed task decomposition and monitoring capabilities supported by robotic execution languages like RAPS(Firby 1995), TCA(Simmons 1994) and ESL(Gat 1996).

Example. We will consider the Autonavigation system on the spacecraft Deep Space 1. This system is used on the spacecraft once a week to perform small course corrections. It takes pictures of three asteroids, and uses the difference between their actual locations from their projected locations to determine the course error. This is then used by another system to determine a new course. The plant model for this consists of models of the camera, the thrusters which turn the spacecraft, and the memory system which stores the pictures. Each of these has various failure modes, which are described in the models. The control program issues commands to the thrusters to turn to each asteroid in turn, to the camera to take pictures, and to the memory to store the pictures. If there is any error, it stops the sequence and takes a recovery action.

We will need to make sure that RMPL is sufficiently expressive to model this system. This behavior draws upon many of the constructs supported by Livingstone—qualitative interactions, multiple modes, concurrent operation and probabilistic transitions. In addition the model must embody a richer set of behaviors that cannot be expressed within Livingstone's modeling language— preemption, forking concurrent processes and iteration. In the next few sections we will show how these are combined into a succinct language.

Hierarchic Constraint Automata

RMPL programs may be viewed as specifications of partially observable Markov processes, that is probabilistic automata with partial observability. While Markov processes offer a natural way of thinking about reactive systems, as a direct encoding they are notoriously intractable. To develop an expressive, yet compact encoding we introduce five key attributes. First, transitions are probabilistic, with associated costs. Second, the Markov process is factored into a set of concurrent automata. Third, each state is labeled with a constraint that holds whenever the automaton marks that state. This allows an efficient, intentional encoding of co-temporal processes, such as fluid flows. Fourth, automata are arranged in a hierarchy – the state of an automaton may itself be an automaton, which is activated by its parent. This enables the initiation and termination of more complex concurrent and sequential behaviors than were possible with the modeling language used in Livingstone. Finally, each transition may have multiple targets, allowing an automaton to be in several states simultaneously, enabling a compact representation of recursive behaviors.

These attributes are a synthesis of representations from several areas of computation. The first attribute comes from the area of Markov processes, and is essential for tasks like stochastic control or failure analysis and repair. The second and third attributes are prevalent in areas like digital systems and qualitative mod-

eling. The fourth and fifth are prevalent in the field of synchronous programming, and form the basis for reactive languages like Esterel(Berry & Gonthier 1992), Lustre(Halbwachs, Caspi, & Pilaud 1991), Signal(Guernic *et al.* 1991) and State Charts(Harel 1987). Together they allow modeling of complex systems that involve software, digital hardware and continuous processes.

Hierarchic constraint automata (HCA) incorporate each of these attributes. An HCA models physical processes with changing interactions by enabling and disabling constraints within a constraint store (e.g., a valve opening causes fuel to flow to an engine). Transitions between successive states are then conditioned on constraints entailed by that store (e.g., the presence or absence of acceleration).

A constraint system $(D, \models)$ is a set of tokens D, closed under conjunction, together with an entailment relation $\models \subseteq D \times D$ (Saraswat 1992). The relation $\models$ satisfies the standard rules for conjunction[†].

RBurton, uses *propositional state logic* as its constraint system. Each proposition is an assignment $x_i = v_{ij}$, where variable x_i ranges over a finite domain $\mathcal{D}(x_i)$. Propositions are composed into formulas using the standard logical connectives – and ($\wedge$), or ($\vee$) and not ($\neg$). If a variable can take on multiple values, then $x_i = v_{ij}$ is replaced with $v_{ij} \in x_i$.

Definition 1 *A deterministic* hierarchical constraint automaton $\mathcal{S}$ *is a 5-tuple* $\langle \Sigma, \Theta, \Pi, \mathcal{C}_P, \mathcal{T}_P \rangle$, *where:*

- Σ *is a set of* states, *partitioned into* primitive states Σ_p *and* composite states Σ_c. *Each composite state itself is a hierarchical constraint automaton.*
- $\Theta \subset \Sigma$ *is a set of* start states.
- Π *is a set of variables with each* $x_i \in \Pi$ *ranging over a finite domain* $\mathcal{D}[x_i]$. $\mathcal{C}[\Pi]$ *denotes the set of all finite domain constraints over* Π.
- $\mathcal{C}_P : \Sigma_p \rightarrow \mathcal{C}[\Pi]$, *associates with each primitive state* s_i *a finite domain constraint* $\mathcal{C}_P(s_i)$ *that holds whenever* s_i *is marked.*
- $\mathcal{T}_P : \Sigma_p \times \mathcal{C}[\Pi] \rightarrow 2^{\Sigma}$ *associates with each primitive state* s_i *a transition function* $\mathcal{T}_P(s_i)$. *Each* $\mathcal{T}_P(s_i) : \mathcal{C}[\Pi] \rightarrow 2^{\Sigma}$, *specifies a set of states to be marked at time* $t+1$, *given assignments to* Π *at time* t.

Simulating Deterministic HCA

A *full marking* of an automaton is a subset of states of an automaton, together with the start states of any composite states in the marking. This is computed recursively from an initial set of states M using $\mathcal{M}_F(M) = M \cup \bigcup\{\mathcal{M}_F(\Theta(s)) \mid s \in M, s \text{ composite}\}$.

Given a full marking M on an automaton A, the function $Step(A, M)$ computes a new full marking corresponding to the automaton transitioning one time step.

[†] 1) $a \models a$ (identity)
2) $a \wedge b \models a$ and $a \wedge b \models b$ ($\wedge$ elimination)
3) $a \models b$ and $b \wedge c \models d$ implies $a \wedge c \models d$ (cut)
4) $a \models b$ and $a \models c$ implies $a \models b \wedge c$ ($\wedge$ introduction)

Step(A, M)::
1. $M1 := \{s \in M \mid s \text{ primitive}\}$
2. $C := \bigwedge_{s \in M1} \mathcal{C}_P(s)$
3. $M2 := \bigcup_{s \in M1} \mathcal{T}_P(s, C)$
4. return $\mathcal{M}_F(M2)$

Step 1 throws away any composite marked states, they are uninteresting as they lack associated constraints or transitions. Step 2 computes the conjunction of the constraints implied by all the primitive states in M. Step 3 computes for each primitive state the set of states it transitions to after one time step. In step 4, applying $\mathcal{M}_F$ to the union of these states marks the start states of any composite state. The result is the full marking for the next time step.

A *trajectory* of an automaton A is a finite or infinite sequence of markings $m_0, m_1, \ldots$, such that m_0 is the initial marking $\mathcal{M}_F(\Theta(A))$, and for each $i \geq 0$, $m_{i+1} = Step(A, m_i)$.

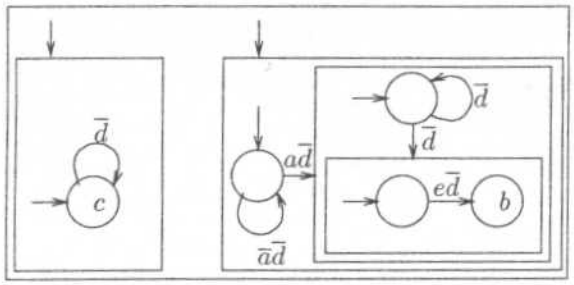
Elaborating on step 3, we represent the transition function for each primitive state $\mathcal{T}_P(s)$ as a set of pairs (l_i, s_i), where $s_i \in \Sigma$, and l_i is a set of labels of the form $\models c$ or $\not\models c$, for some $c \in \mathcal{C}[\Pi]$. This is the traditional representation of transitions, as a labeled arc in a graph. If the automaton is in state s, then at the next instant it will go to *all* states s_i whose label l_i is entailed by constraints C, as computed in the second step of the algorithm. l_i is said to be entailed by C, written $C \models l_i$, if $\forall \models c \in l_i . C \models c$, and for each $\not\models c \in l_i . C \not\models c$. It is straightforward to translate this representation into our formal representation: $\mathcal{T}_P(s, C) = \{s_i \mid C \models l_i\}$.

Two properties of these transitions are distinctive: Transitions are conditional on what can be deduced, not just what is explicitly assigned, and transitions are enabled based on lack of information.

Step provides a deterministic simulator for the plant, when applied to an HCA that specifies a plant model. Alternatively Step provides a deterministic version of the control sequencer for RBurton, by placing appropriate restrictions on the control HCA. Constraints attached to primitive states on this HCA are restricted to control assignments, while transition labels are conditioned on the external goals and the estimated current state. The set of active constraints collected from marked states during step 2 of the algorithm is then the set of control actions to be output to the plant.

A Simple Example We illustrate HCA with a simple automaton. In this picture c represents a constraint, start states of an automaton are marked with arrows, and all transitions are labeled. For convenience we use c to denote the label $\models c$, and $\bar{c}$ to denote the label $\not\models c$. Circles represent primitive states, while rectangles represent composite states.

The automaton has two start states, both of which are composite. Every transition is labeled $\not\models d$, hence all transitions are disabled and the automaton is preempted whenever d becomes true. The first state has one primitive state, which asserts the constraint c. If d does not hold, then it goes back to itself — thus it



repeatedly asserts c until d becomes true. The second automaton has a primitive start state. Once again, at anytime if d becomes true, the entire automaton will immediately terminate. Otherwise it waits until a becomes true, and then goes to its second state, which is composite. This automaton has one start state, which it repeats at every time instant until d holds. In addition, it starts another automaton, which checks if e holds, and if true generates b in the next state. Thus, the behavior of the overall automaton is as follows: it starts asserting c at every time instant. If a becomes true, then at every instant thereafter it checks if e is true, and asserts b in the succeeding instant. Throughout it watches for d to become true, and if so halts. An RMPL program that produces an equivalent automaton is:

do
 (**always** c,
 when a **donext always if** e **thennext** b)
watching d

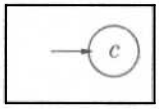
The combinators of this program are developed during the remainder of this paper.

RMPL: Primitive Combinators

We now present the syntax for the reactive model-based programming language. Our preferred approach is to introduce a minimum set of primitives, used to construct programs — each primitive that we add to the language is driven by a desired feature. We then define on top of these primitives a variety of program combinators, such as those used in the simple example above, that make the language usable. The primitives are driven by the need to write reactive control software in the language, as well as to model physical systems. As mentioned earlier, to write reactive control programs we require combinators for preemption, conditional branching and iteration. For modeling hardware, we require constructs for representing constraints and probability. Finally we need logical concurrency to be able to compose models and programs together.

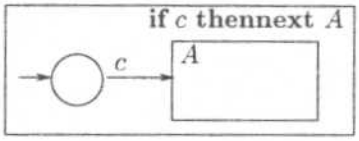
As we introduce each primitive we show how to construct its corresponding automaton. In these definitions lower case letters, like c, denote constraints, while upper case letters, like A and B, denote automata. The term "theory" refers to the set of all constraints associated with marked primitive states at some time point.

c. This program asserts that constraint c is true at the initial instant of time. This construct is used to represent co-temporal interactions, such as a qualitative constraint between fluid flow and pressure. The automaton for it is:

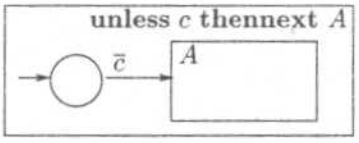


Note that the start state in this automaton has no exit transitions, so after this automaton asserts c in the first time instant it terminates.

if c **thennext** A. This program starts behaving like A in the next instant if the current theory entails c. This is the basic conditional branch construct. Given the automaton for A, we construct an automaton for **if** c **thennext** A by adding a new start state, and going from this state to A if c is entailed.

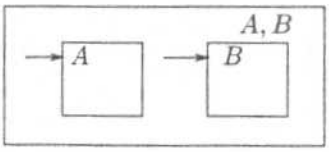


unless c **thennext** A. This program executes A in the next instant if the current theory does *not* entail c. The automaton for this is similar to the automaton for **if** c **thennext** A. This is the basic construct for building preemption constructs — it is the only one that introduces conditions $\not\models c$ (written in the automaton as $\bar{c}$). This introduces non-monotonicity, but since the non-monotonic conditions trigger effects in the next instant, the logic is stratified and monotonic in each state. This avoids the kinds of causal paradoxes possible in languages like Esterel[‡].



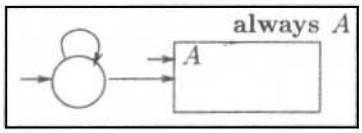
We also allow generalized sequences for **if** ... **then** and **unless** ... **then**, terminated with **thennext**. (*e.g.* **if** c **then unless** d **thennext** A). The compilation to automata proceeds exactly as above.

A, B. This is the parallel composition of two automata, and is the basic construct for introducing concurrency. The composite automaton has two start states, given by the two automata for A and B.



[‡]Esterel allows programs like **unless** c **then** c, which have no behavior. These are rejected by its compiler.

always A. This program starts a new copy of A at each instant of time — this is the only iteration construct needed. The automaton is produced by marking A as a start state and by introducing an additional new start state. This state has the responsibility of initiating A during every time step after the first. A transition back to itself ensures that this state is always marked. A second transition to A puts a new mark on the start state of A at every next step, each time invoking a virtual copy of A. The ability of an automaton to have multiple states marked simultaneously is key to this novel encoding, which avoids requiring explicit copies of A.



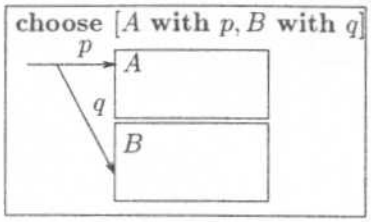
Adding Uncertainty to RMPL

The presentation has concentrated thus far on a deterministic language and an algorithm for deterministically executing hierarchical constraint automata. This can be used to simulate a deterministic plant or to generate deterministic plant control sequences. However physical plant models are frequently uncertain, thus forcing us to build probabilistic models. The plant's observables are used to predict its internal state, and to determine when it deviates from the intended effect. Uncertainty is modeled by introducing transition probabilities, turning the plant into a partially observable Markov process. The efficient estimation of the internal state for complex systems is notoriously difficult.

An efficient estimate of the plant's possible states (*the belief state*) is enabled through the compact encoding of the plant's model in terms of hierarchical constraint automata. This estimate is used to guide the evaluation of the control program at each time tick. To express probabilistic knowledge into RMPL we introduce the probabilistic combinator **choose** :

choose $[A$ **with** p, B **with** $q]$. This combinator reduces to A with probability p, to B with probability q, and so on. In order to ensure that the current theory does not depend upon the probabilistic choices made in the current state, we make the following restriction — all assertions of constraints in A and B must be in the scope of a **next**. This restriction ensures that no constraints are associated with the start states of A and B (technically the attached constraint is "true"), and thus the probabilities are associated only with transitions. The corresponding automaton is encoded with a single probabilistic start transition, which allows us to choose between A and B.

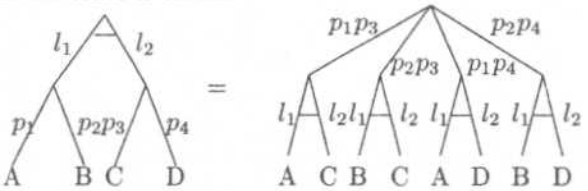
To incorporate probabilistic transitions into HCA we change the definition of $\mathcal{T}_P$. Recall for deterministic HCA that $\mathcal{T}_P(s_i)$ denotes a single transition function. For probabilistic HCA $\mathcal{T}_P(s_i)$ denotes a distribution over transition functions $\mathcal{T}_P{}^j(s_i)$, whose probabilities $\mathbf{P}(\mathcal{T}_P{}^j(s_i))$ sum to 1.



$\mathcal{T}_P(s_i)$ is encoded as a probabilistic, *AND-OR* tree. This supports a simple transformation of nested **choose** combinators to probabilistic HCA. Each leaf of this tree is labeled with a set of one or more *target states* in Σ, which the automaton transitions to in the next time tick.

The branches $a_i \rightarrow b_{ij}$ of a probabilistic *OR* node a_i represent a distribution over a disjoint set of alternatives, and are labeled with conditional probabilities $\mathbf{P}[b_{ij} \mid a_i]$. The probability of branches emanating from each a_i sum to unity.

The branches of a deterministic *AND* node represent an inclusive set of choices. Each branch is labeled by a set of conditions l_{ij} of the form $\models \phi$ or $\not\models \phi$, where ϕ is any formula in propositional state logic over variables Π. Every branch whose conditions are satisfied by the current state is taken.



Each AND-OR tree is compiled into a two level tree (shown above), with the root node being a probabilistic OR, and its children being deterministic ANDs. Compilation is performed using distributivity, as shown above, and commutativity, which allows adjacent *AND* nodes to be merged, by taking conjunctions of labels, and adjacent *OR* nodes to be merged, by taking products of probabilities.

This two level tree is a direct encoding of $\mathcal{T}_P(s_i)$. Each AND node represents one of the transition functions $\mathcal{T}_P{}^j(s_i)$, while the probability on the OR branch, terminating on this AND node, denotes $\mathbf{P}(\mathcal{T}_P{}^j(s_i))$.

RBurton: State Estimation

To implement belief state update recall that a probabilistic HCA encodes a partially observable Markov process. A partially observable Markov process can be described as a tuple $\langle \Sigma, \mathcal{M}, \mathcal{O}, \mathbf{P}_\mathcal{T}, \mathbf{P}_\mathcal{O} \rangle$. Σ, $\mathcal{M}$ and $\mathcal{O}$ denote finite sets of *feasible states* s_i, *control actions* μ_i, and *observations* o_i respectively. The *state transition function*, $\mathbf{P}_\mathcal{T}[s_i{}^{(t)}, \mu_i{}^{(t)} \mapsto s_i{}^{(t+1)}]$ denotes the probability that $s_i{}^{(t+1)}$ is the next state, given current state $s_i{}^{(t)}$ and control action $\mu_i{}^{(t)}$ at time t. The *observation function*, $\mathbf{P}_\mathcal{O}[s_i{}^{(t)} \mapsto o_i{}^{(t)}]$ denotes the probability that

$o_i^{(t)}$ is observed, given state $s_i^{(t)}$ at time t.

RBurton incrementally updates the plant belief state, conditioned on each control action sent and each observation received, respectively:

$$\sigma^{(\bullet t+1)}[s_i] \equiv \mathbf{P}[s_i^{(t+1)} \mid o_{v_0}^{(0)}, \ldots, o_{v_t}^{(t)}, \mu_{v_0}^{(0)} \ldots \mu_{v_t}^{(t)}]$$
$$\sigma^{(t+1\bullet)}[s_i] \equiv \mathbf{P}[s_i^{(t+1)} \mid o_{v_0}^{(0)}, \ldots, o_{v_t+1}^{(t+1)}, \mu_{v_0}^{(0)} \ldots \mu_{v_t}^{(t)}]$$

Exploiting the Markov property, the belief state at time $t+1$ is computed from the belief state and control actions at time t and observations at $t+1$ using the standard equations:

$$\sigma^{(\bullet t+1)}[s_i] = \sum_{j=1}^{n} \sigma^{(t\bullet)}[s_j]\mathbf{P}_{\mathcal{T}}[s_i, \mu_i \mapsto s_j]$$

$$\sigma^{(t+1\bullet)}[s_i] = \sigma^{(\bullet t+1)}[s_i]\frac{\mathbf{P}_{\mathcal{O}}[s_i \mapsto o_k]}{\sum_{j=1}^{n} \sigma^{(\bullet t+1)}[s_j]\mathbf{P}_{\mathcal{O}}[s_j \mapsto o_k]}$$

To calculate $\mathbf{P}_{\mathcal{T}}$ recall that a transition $\mathcal{T}$ is composed of a set of primitive transitions, one for each marked primitive state. Assuming conditional independence of primitive transition probabilities, given the current marking, the combined probability of each set is the product of the primitive transition probabilities of the set. This is analogous to the various independence of failure assumptions exploited by systems like GDE(de Kleer & Williams 1987), Sherlock(de Kleer & Williams 1989) and Livingstone. However unlike these systems, multiple sets of transitions may go to the same target marking. This is a consequence of the fact that in an HCA primitive states have multiple next states. Hence the transition probabilities for all transitions going to the same target must be summed according to the above equation for $\sigma^{(\bullet t+1)}[s_i]$.

Given $\mathbf{P}_{\mathcal{T}}$, the belief update algorithm for $\sigma^{(\bullet t+1)}[s_i]$ is a modified version of the Step algorithm presented earlier. This new version of Step returns a set of markings, each with its own probability. Step 3a builds the sets of possible primitive transitions — here the product is a Cartesian product of sets. Step 3b computes the combined next state marking and transition probability of each set. Step 3c sums the probability of all composite transitions with the same target:

Step$_P(A, M)$::
1. $M1 := \{s \in M \mid s \text{ primitive}\}$
2. $C := \bigwedge_{s \in M1} \mathcal{C}_P(s)$
3a. $M2a := \prod_{s \in M1} \mathcal{T}_P(s, C)$
3b. $M2b := \{(\mathcal{M}_F(\bigcup_{i=1}^{n} S_i), \prod_{i=1}^{n} p_i) \mid \langle (S_1, p_1), \ldots, (S_n, p_n) \rangle \in M2a\}$
3c. $M2 := \{(S, \sum_{(S,p) \in M2b} p) \mid (S, \_) \in M2b\}$
4. return $M2$

The best first enumeration algorithms developed for Sherlock and Livingstone are directly used by RBurton to generate the composite transitions in step 3a and b in order from most to least likely. However, since the correspondence between transitions and next states is many to one, there is no guarantee that the belief states are enumerated in decreasing order.

Instead we assume that most of the probability density resides in the few leading candidate transition sets. Hence a best first enumeration of the few leading transition sets will quickly lead to a reasonable approximation. We enumerate transitions in decreasing order until most of the probability density space is covered (e.g., 95%), and then perform step 3c to merge the results.

Computing $\sigma^{(t+1\bullet)}[s_i]$ requires $\mathbf{P}_{\mathcal{O}}[s_i^{(t)} \mapsto o_i^{(t)}]$. $\mathbf{P}_{\mathcal{O}}$ is computed using the standard approach in model-based reasoning, first introduced within the GDE system. For each variable assignment in each new observation, RBurton uses the model, current state and previous observation to predict or refute this assignment, giving it probability 1 or 0 respectively. If no prediction is made, then a prior distribution on observables is assumed (e.g., $1/n$ for n possible values).

RBurton: Mode Reconfiguration and Sequencing

A full decision theoretic executive that maximizes expected reward using HCA is well beyond the scope of this paper. However, a simplified executive that does task decomposition based on immediate rewards can be easily constructed by a simple extension to the above model. Thus RBurton maximizes immediate reward under the assumption that the most likely estimated state is correct. We further assume that rewards are additive. The hierarchical automaton provides a way of structuring tasks, subtasks and solution methods.

We restrict the constraints c of a control program to plant control assignments. In addition, to support selection of methods for tasks, we replace the probabilistic combinator **choose** with an analogous combinator based on reward:

choosereward $[A$ **with** p, B **with** $q]$. This combinator reduces to A with reward p, to B with reward q, and so on. **choosereward** has restrictions analogous to **choose** that associate rewards only to expressions containing **next** .

The AND-OR Tree formed by nested applications of **choosereward** is analogous to **choose** . The tree is reduced in a similar manner, except that rewards are added while probabilities are multiplied.

Control sequence generation again uses a variation of Step. For step 3 of this algorithm a best first enumeration algorithm is given the sets of enabled transitions from each primitive state that is marked in the most likely current marking. During the enumeration it must rule out any sets of transitions that lead to an inconsistent (conflicting) control assignment. It then returns the set of transitions that maximize combined reward. This is analogous in RAPS(Firby 1995) to selecting applicable methods based on priority numbers.

Extending RMPL: Definable operators

Given the basic operators defined earlier, we can define a variety of common language constructs, making

the task of programming in reactive MPL considerably easier. Common constructs in RMPL include recursion, next, sequencing and preemption.

Recursion and procedure definitions. Given a declaration $P :: A[P]$, where A may contain occurrences of procedure name P, we replace it by **always if** p **then** $A[p/P]$. At each time tick this looks to see if p is asserted (corresponding to P being invoked), and if so starts A.

next A. This is simply **if** *true* **thennext** A. We can also define **if** c **thennext** A **elsenext** B as **if** c **thennext** A, **unless** c **thennext** B.

if c **then** A. This construct has the effect of starting A at the time instant in which c becomes true. It can be defined as follows, where the expression to the left of the equality is replaced with the expression on the right (note that sequences of **if** ... **then** and **unless** ... **then**, terminated with **thennext** are OK):

if c **then** $d = c \rightarrow d$
if c **then always** $A =$
  if c **then** A, **if** c **thennext always** A
if c **then** $(A, B) =$
  if c **then** A, **if** c **then** B
if c **then choose** $[A$ **with** p, B **with** $q] =$
  choose [**if** c **then** A **with** p, **if** c **then** B **with** q]

A; B. This does sequential composition of A and B. It keeps doing A until A is finished. Then it starts B. It can be written in terms of the other constructs by detecting the termination of A by a proposition, and using that to trigger B. RMPL detects the termination of A by a case analysis of the structure of A (see (Fromherz, Gupta, & Saraswat 1997) for details).

do A **watching** c. This is a weak preemption operator. It executes A, but if c becomes true in any time instant, it terminates execution of A in the next instant. The automaton for this is derived from the automaton for A by adding the label $\not\models c$ on all transitions in A.

suspend A **on** c **reactivate on** d. This is like the "$Control - Z, fg$" pair of Unix — it suspends the process when c becomes true, and restarts it from the same point when d becomes true.

when c **donext** A. This starts A at the instant after the first one in which c becomes true. It is a temporally extended version of **if** c **thennext** A. It can be coded using recursion and **if** c **thennext** A.

DS1 Optical Navigation Example

To illustrate RMPL further, we give some fragments of the RMPL models for the Autonav example. MICAS is a miniature camera on DS1.

```
MICAS :: always {
  choose {
    {
      if MICASon then
        if TurnMicasOff thennext MICASoff
        elsenext MICASon,
      if MICASoff then ...,
      if MICASfail then ...,
    } with 0.99,
    next MICASfail with 0.01
  }
}
```

This model shows how probability is used — at each step, MICAS will fail with probability 0.01. Otherwise it will continue to exhibit normal behavior, during which it can respond to commands. Probability can also be used to simulate the effects of reseting a failed MICAS — a reset corrects the failure with some probability.

AutoNav is the control routine that performs the course correction.

```
AutoNav() :: {
  TurnMicasOn,
  if IPSon thennext SwitchIPSStandBy,
  do when IPSstandby ∧ MICASon donext {
    TakePicture(1);...
    {
      TurnMicasOff,
      OpticalNavigation()
    }
  } watching MICASfail ∨ OpticalNavError,
  when MICASfail donext {MICASReset, AutoNav()},
  when OpticalNavError donext AutoNavFailed
}
```

This routine shows how iteration and preemption are used. The preemption operator **do** ... **watching** aborts the current program in the face of an error, and restarts the program. The modular nature of the preemption is particularly useful here — the program inside the **do** ... **watching** can be written without any concern for failures, which are captured by the preemption operation, somewhat like exceptions in Java.

Discussion and Related Work

The RMPL compiler is written in C, and generates hierarchical constraint automata as its target. This supports all primitive combinators and a variety of defined combinators. RBurton is written in Lisp, and builds upon the best-first enumeration code at the heart of the Livingstone system. In addition the language is sufficiently expressive and compact to support the full DS1 spacecraft models developed for Livingstone. RBurton's behavior is equivalent to Livingstone for those examples.

Turning to related work, RMPL synthesizes ideas underlying constraint-based modeling, synchronous programming languages and Markov processes. Synchronous programming languages (Halbwachs 1993; Berry & Gonthier 1992; Halbwachs, Caspi, & Pilaud 1991; Guernic *et al.* 1991; Harel 1987; Saraswat, Jagadeesan, & Gupta 1996) were developed for writing

control code for reactive systems. They are based on the Perfect Synchrony Hypothesis — a program reacts instantaneously to its inputs. Synchronous programming languages exhibit logical concurrency, orthogonal preemption, multiform time and determinacy, which Berry has convincingly argued are necessary characteristics for reactive programming. RMPL is a synchronous language, and satisfies all these characteristics.

In addition, RMPL is distinguished by the adoption of Markov processes as its underlying model, its treatment of partial observability and its extensive use of contraint modeling to observe hidden state. This provides a rich language for continuous process, failure, uncertainty and repair.

As previously discussed, RMPL and RBurton overlap substantially with AI robotic execution languages RAPS, ESL and TCA. For example, method selection, monitoring, preemption and concurrent execution are core elements of these languages, shared with RMPL.

One key difference is that RMPL's constructs fully cover synchronous programing, hence moving towards a unification of the executive with the underlying real-time language. In addition RBurton's deductive monitoring capability handles a rich set of software/hardware models that go well beyond those handled by systems like Livingstone. This moves execution languages towards a unification with model-based, deductive monitoring.

Finally, note that hierarchical state diagrams, like State Charts(Harel 1987), are becoming common tools for system engineers to write real-time specifications. These specifications are naturally expressed within RMPL, due to RMPL's simple correspondence with hierarchical constraint automata, which are closely related to state charts. Together this offers a four way unification between synchronous programming, robotic execution, model-based autonomy and real-time specification, – a significant step towards our original goal.

Nevertheless substantial work remains. Many execution and control capabilities key to highly autonomous systems fall well outside the scope of RMPL and RBurton. For example, RMPL has no construct for expressing metric time. Hence RBurton cannot execute or monitor temporal plans without the aid of an executive like RAPS or Remote Agent's Exec. In addition, outside of monitoring, RBurton does not employ any deduction or planning during control sequence generation. Unifying the kinds of sequence generation capabilties that are the hallmark of systems like HSTS(Muscettola 1994) and Burton(Williams & Nayak 1997), requires significant research.

References

Benveniste, A., and Berry, G., eds. 1991. *Another Look at Real-time Systems*, volume 79:9.

Berry, G., and Gonthier, G. 1992. The ESTEREL programming language: Design, semantics and implementation. *Science of Computer Programming* 19(2):87 – 152.

Berry, G. 1989. Real-time programming: General purpose or special-purpose languages. In Ritter, G., ed., *Information Processing 89*, 11 – 17. Elsevier.

de Kleer, J., and Williams, B. C. 1987. Diagnosing multiple faults. *Artificial Intelligence* 32(1):97–130.

de Kleer, J., and Williams, B. C. 1989. Diagnosis with behavioral modes. In *Proceedings of IJCAI-89*, 1324–1330.

Firby, R. J. 1995. The RAP language manual. Animate Agent Project Working Note AAP-6, University of Chicago.

Fromherz, M.; Gupta, V.; and Saraswat, V. 1997. cc – A generic framework for domain specific languges. In *POPL Workshop on Domain Specific Languages.*

Gat, E. 1996. Esl: A language for supporting robust plan execution in embedded autonomous agents. In *Proceedings of the 1996 AAAI Fall Symposium on Plan Execution.*

Guernic, P. L.; Borgne, M. L.; Gauthier, T.; and Maire, C. L. 1991. Programming real time applications with SIGNAL. In *Proceedings of the IEEE* (1991) 1321–1336.

Halbwachs, N.; Caspi, P.; and Pilaud, D. 1991. The synchronous programming language LUSTRE. In *Proceedings of the IEEE* (1991) 1305–1320.

Halbwachs, N. 1993. *Synchronous programming of reactive systems.* Series in Engineering and Computer Science. Kluwer Academic.

Harel, D., and Pnueli, A. 1985. *Logics and Models of Concurrent Systems*, volume 13. NATO Advanced Study Institute. chapter On the development of reactive systems, 471–498.

Harel, D. 1987. Statecharts: A visual approach to complex systems. *Science of Computer Programming* 8:231 – 274.

Muscettola, N.; Nayak, P. P.; Pell, B.; and Williams, B. C. 1999. The new millennium remote agent: To boldly go where no ai system has gone before. *Artificial Intelligence* 100.

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

Saraswat, V. A.; Jagadeesan, R.; and Gupta, V. 1996. Timed Default Concurrent Constraint Programming. *J of Symbolic Computation* 22(5-6):475–520.

Saraswat, V. A. 1992. The Category of Constraint Systems is Cartesian-closed. In *Proc. 7th IEEE Symp. on Logic in Computer Science, Santa Cruz.*

Simmons, R. 1994. Structured control for autonomous robots. *IEEE Transactions on Robotics and Automation* 10(1).

Williams, B. C., and Nayak, P. P. 1996. A model-based approach to reactive self-configuring systems. In *Proceedings of AAAI-96*, 971–978.

Williams, B. C., and Nayak, P. P. 1997. A reactive planner for a model-based executive. In *Proceedings of IJCAI-97.*