Mode Estimation of Model-based Programs: Monitoring Systems with Complex Behavior

Brian C. Williams and **Seung Chung**
Massachusetts Institute of Technology
77 Massachusetts Ave. Rm. 37-381
Cambridge, MA 02139 USA
{williams,chung}@mit.edu

Vineet Gupta
PurpleYogi, Inc.
201 Ravendale Drive
Mountain View CA 94043
vineet@purpleyogi.com

Abstract

Deductive mode-estimation has become an essential component of robotic space systems, like NASA's deep space probes. Future robots will serve as components of large robotic networks. Monitoring these networks will require modeling languages and estimators that handle the sophisticated behaviors of robotic components. This paper introduces *RMPL*, a rich modeling language that combines reactive programming constructs with probabilistic, constraint-based modeling, and that offers a simple semantics in terms of hidden Markov models (HMMs). To support efficient real-time deduction, we translate RMPL models into a compact encoding of HMMs called *probabilistic hierarchical constraint automata (PHCA)*. Finally, we use these models to track a system's most likely states by extending traditional HMM belief update.

1 Introduction

Highly autonomous systems are being developed, such as NASA's Deep Space One probe (DS-1) and the X-34 Reusable launch vehicle, that involve sophisticated model-based planning and mode-estimation capabilities to support autonomous commanding, monitoring and diagnosis. Given an observation sequence, a mode estimator, such as Livingstone [Williams and Nayak, 1996], incrementally tracks the most likely state trajectories of a system, in terms of the correct or faulty modes of every component.

A recent trend is to aggregate autonomous systems into robotic networks, for example, that create multi-spacecraft telescopes, perform coordinated Mars exploration, or perform multi vehicle search and rescue. Novel model-based methods need to be developed to monitor and coordinate these complex systems.

An example of a complex device is DS-1, which flies by an asteroid and comet using ion propulsion. DS-1's basic functions include weekly course correction (called *optical navigation*), thrusting along a desired trajectory, taking science readings and transferring data to earth. Each function involves a complex coordination between software and hardware. For example, optical navigation (OPNAV) works by taking pictures of three asteroids, and by using the difference between actual and projected locations to determine the course error. OPNAV first shuts down the Ion engine and prepares its camera concurrently. It then uses the thrusters to turn to each of three asteroids, uses the camera to take a picture of each, and stores each picture on disk. The three images are then read, processed and a course correction is computed. One of the more subtle failures that OPNAV may experience is a corrupted camera image. The camera generates a faulty image, which is stored on disk. At some later time the image is read, processed, and only then is the failure detected. A monitoring system must be able to estimate this event sequence based on the delayed symptom.

Diagnosing the OPNAV failure requires tracking a trajectory that reflects the above description. Identifying this trajectory goes well beyond Livingstone's abilities. Livingstone, like most diagnostic systems, focuses on monitoring and diagnosing networks whose components, such as valves and bus controllers, have simple behaviors. However, the above trajectory spends most of its time wending its way through software functions. DS-1 is an instance of modern embedded systems whose components involve a mix of hardware, computation and software. Robotic networks extend this trend to component behaviors that are particularly sophisticated.

This paper addresses the challenge of modeling and monitoring systems composed of these complex components. We introduce a unified language that can express a rich set of mixed hardware *and* software behaviors (the *Reactive Model-based Programming Language [RMPL]*). RMPL merges constructs from synchronous programming languages, qualitative modeling, Markov models and constraint programming. Synchronous, embedded programming offers a class of languages developed for writing control programs for reactive systems [Benveniste and Berry, 1991; Saraswat *et al.*, 1996] — logical concurrency, preemption and executable specifications. Markov models and constraint-based modeling[Williams and Nayak, 1996] offer rich languages for describing uncertainty and continuous processes at the qualitative level.

Given an RMPL model, we frame the problem of monitoring robotic components as a variant of *belief update* on a *hidden Markov model (HMM)*, where the HMM of the system is described in RMPL. A key issue is the potentially enormous state space of RMPL models. We address this by introducing a hierachical, constraint-based encoding of an HMM, called a probabilistic, hierarchical, constraint automata (PHCA). Next we show how RMPL models can be compiled to equivalent PHCAs. Finally, we demonstrate one approach in which RMPL belief update can be performed by operating directly on the compact PHCA encoding.

2 HMMs and Belief Update

The theory of HMMs offers a versatile tool for framing hidden state interpretation problems, including data transmission, speech and handwriting recognition, and genome sequencing. This section reviews HMMs and state estimation through belief update.

An HMM is described by a tuple $\langle \Sigma, \mathcal{O}, \mathbf{P}_\Theta, \mathbf{P}_\mathcal{T}, \mathbf{P}_\mathcal{O} \rangle$. Σ and $\mathcal{O}$ denote finite sets of *feasible states* s_i and *observations* o_i. The *initial state function*, $\mathbf{P}_\Theta[{s_i}^{(0)}]$, denotes the probability that s_i is the initial state. The *state transition function*, $\mathbf{P}_\mathcal{T}[{s_i}^{(t)} \mapsto {s_i}^{(t+1)}]$, denotes the conditional probability that ${s_i}^{(t+1)}$ is the next state, given current state ${s_i}^{(t)}$ at time t. The *observation function*, $\mathbf{P}_\mathcal{O}[{s_i}^{(t)} \mapsto {o_i}^{(t)}]$ denotes the conditional probability that ${o_i}^{(t)}$ is observed, given state ${s_i}^{(t)}$.

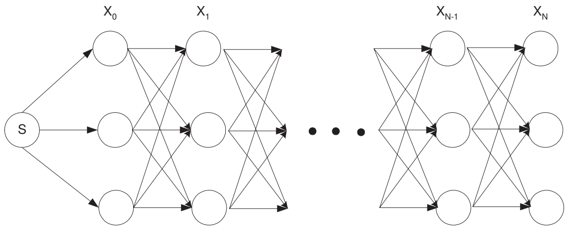
Belief update incrementally computes the current belief state, that is, the likelihood that the system is in any state s_i, conditioned on each control action performed and observation received, respectively:

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

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. For simplicity, control actions are made implicit within $\mathbf{P}_\mathcal{T}$:

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

The space of possible trajectories of an HMM can be visualized using a *Trellis diagram*, which enumerates all possible states at each time step and all transitions between states at adjacent times. Belief update associates a probability to each state in the graph.



3 Design Desiderata for RMPL

Returning to our example, OPNAV is best expressed at top-level as a program:

```
OpNav() :: {
  TurnCameraOn,
  if EngineOn  thennext SwitchEngineStandBy,
  do
    when EngineStandby ∧ CameraOn donext {
      TakePicture(1);
      TakePicture(2);
      TakePicture(3);
      {
        TurnCameraOff,
        ComputeCorrection()
      }
    } watching PictureError ∨ OpticalNavError,
    when OpticalNavError donext OpNav(),
    when PictureError donext OpNavFailed
  }
```

In this program comma delimits parallel processes and semicolon delimits sequential processes.

OPNAV highlights four key design features for RMPL. First, the program exploits full concurrency, by intermingling sequential and parallel threads of execution. For example, the camera is turned on and the engine is turned off in parallel, while pictures are taken serially. Second, it involves conditional execution, such as switching to standby if the engine is on. Third, it involves iteration; for example, "**when** Engine Standby ... **donext** ... " says to iteratively test to see if the engine is in standby and if so proceed. Fourth, the program involves preemption; for example, "**do** ... **watching** " says to perfom a task, but to interrupt it as soon as the watch condition is satisfied. Subroutines used by OpNav, such as TakePicture, exploit similar features.

OpNav also relies on hardware behaviors, such as:

```
Camera :: always {
  choose {
    {
      if CameraOn then {
        if TurnCameraOff thennext MicasOff
        elsenext CameraOn,
        if CameraTakePicture thennext CameraDone
      },
      if CameraOff  then
        if TurnCameraOn thennext CameraOn
        elsenext CameraOff,
      if CameraFail  then
        if MicasReset thennext CameraOff
        elsenext CameraFail
    } with0.99,
    next CameraFail with 0.01
  }
}
```

OpNav's tight interaction with hardware makes the overall process stochastic. We add probabilistic execution to our design features to model failures and uncertain outcomes. We add constraints to represent co-temporal interactions between state variables. Summarizing, the key design features of RMPL are full concurrency, conditional execution, iteration, preemption, probabilistic choice, and co-temporal constraint.

4 RMPL: Primitive Combinators

Our preferred approach to developing RMPL is to introduce a minimum set of primitives for constructing programs, where each primitive is driven by one of the six design features of the preceding section. To make the language usable we define on top of these primitives a variety of program combinators, such as those used in the optical navigation example. In the following we use lower case letters, like c, to denote constraints, and upper case letters, like A and B, to denote well-formed RMPL expressions. The term "theory" refers to the set of all constraints that hold at some time point.

c. This program asserts that constraint c is true at the initial instant of time.

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.

unless c **thennext** A. This program executes A in the next instant if the current theory does *not* entail c. This is the basic construct for building preemption constructs. It allows A to proceed as long as some condition is unknown, but stops when the condition is determined.

A, B. This program concurrently executes A and B, and is the basic construct for forking processes.

always A. This program starts a new copy of A at each instant of time, for all time. This is the only iteration construct needed, since finite iterations can be achieved by using **if** or **unless** to terminate an **always** .

choose $[A$ **with** p, B **with** $q]$. This is the basic combinator for expressing probabilistic knowledge. It reduces to program A with probability p, to program B with probability q, and so on. For simplicity we would like to ensure that constraints in the current theory do not depend upon probabilistic choices made in the current state. We achieve this by restricting all constraints asserted within A and B to be within the scope of an **if** ... **next** or **unless** ... **next**.

These six primitive combinators cover the six design features. They have been used to implement a rich set of derived combinators [anonymous] including those in the OpNav example, and most from the Esterel language [Berry and Gonthier, 1992]. The derived operators for OpNav, built from these primitives, is given in Appendix A.

5 Hierarchical, Constraint Automata

To estimate RMPL state trajectories we would like to map the six RMPL combinators to HMMs and then perform belief update. However, while HMMs offer a natural way of thinking about reactive systems, as a direct encoding they are notoriously intractable. One of our key contributions is a representation, called *Probabilistic, Hierarchical, Constraint Automata (PHCA)* that compactly encodes HMMs describing RMPL models.

PHCA extend HMMs by introducing four essential attributes. First, the HMM is factored into a set of concurrently operating automata. Second, each state is labeled with a constraint that holds whenever the automaton marks that state. This allows an efficient encoding of co-temporal processes, such as fluid flows. Third, automata are arranged in a hierarchy – the state of an automaton may itself be an automaton, which is invoked when marked by its parent. This enables the initiation and termination of more complex concurrent and sequential behaviors. Finally, each transition may have multiple targets, allowing an automaton to be in several states simultaneously. This enables a compact representation for recursive behaviors like "always" and "do until".

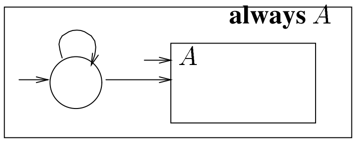
The first two attributes are prevalent in areas like digital systems and qualitative modeling. The third and fourth form the basis for embedded reactive languages like Esterel[Berry and Gonthier, 1992], Lustre[Halbwachs *et al.*, 1991], Signal[Guernic *et al.*, 1991] and State Charts[Harel, 1987]. Together they allow complex systems to be modeled that involve software, digital hardware and continuous processes.

We develop PHCAs by first introducing a deterministic equivalent, and then extending to Markov models. We describe a *deterministic, hierarchical, constraint automaton (HCA)* as a tuple $\langle \Sigma, \Theta, \Pi, \mathcal{O}, \mathcal{C}_P, \mathcal{T}_P \rangle$, where:

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

At any instant t the "state" of an HCA is the set of marked states $m_i{}^{(t)} \subset \Sigma$, called a *marking*. $\mathcal{M}$ denotes the set of possible markings, where $\mathcal{M} = 2^{\Sigma}$.

Consider the combinator **always** A, which maps to:



This automaton starts a new copy of A at each time instant. The states Σ of the automaton consist of primitive state s_{new}, drawn to the left as a circle, and composite state A, drawn to the right as a rectangle. The start states Θ are s_{new} and A, as is indicated by two short arrows.

A PHCA models physical processes with changing interactions by enabling and disabling constraints within a constraint store (e.g., opening a valve causes fuel to flow to an engine). RMPL currently supports *propositional state logic* as its constraint system. In state logic each proposition is an assignment $x_i = v_{ij}$, where variable x_i ranges over a finite domain $\mathcal{D}(x_i)$. Constraints $\mathcal{C}_P$ are indicated by lower case letters, such as c, written in the middle of a primitive state. If no constraint is indicated, the state's constraint is implicitly **True**. In the above example s_{new} implicitly has constraint **True**; other constraints may be hidden within A.

Transitions between successive states are conditioned on constraints entailed by the store (e.g., the presence or absence of acceleration). This allows us to model indirect control and indirect effects. For each primitive state s we represent the transition function $\mathcal{T}_P(s)$ as a set of (*transition*) 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 corresponds to the traditional representation of transitions, as labeled arcs in a graph, where s and s_i are the source and destination of an arc with label l_i. For convenience, in our diagrams we use c to denote the label $\models c$, and $\overline{c}$ to denote the label $\not\models c$. If no label is indicated, it is implicitly $\models$ **True**. The above example has two transitions, both with labels that are implicitly $\models$ **True**.

Our HCA encoding has three key properties that distinguish it from the hierarchical automata employed by reactive embedded languages[Benveniste and Berry, 1991; Harel, 1987]. First, multiple transitions may be simultaneously traversed. This allows an extremely compact encoding of the state of the automaton as a set of markings. Second, transitions are conditioned on what can be deduced, not just what is explicitly assigned. This provides a simple but general mechanism for incorporating constraint systems that reason about indirect effects. Third, transitions are enabled based on lack of information. This allows default executions to be pursued in the absence of better information, enabling advanced preemption constructs.

6 Executing HCA

To execute an automata A, we first initialize it using $m_F(\Theta(A))$, which marks the start states of all its subautomata, and then step it using $Step(A)$, which maps its current marking to a next marking. [1]

> A *trajectory* of automaton A is a sequence of markings ${m_i}^{(0)}, {m_j}^{(1)}, \ldots$ such that ${m_i}^{(0)}$ is the initial marking $m_F(\Theta)$, and for each $l \geq 0$, ${m_i}^{(l+1)} = Step(A, {m_j}^{(l)})$.

Given a set of automata m to be initialized, $m_F(m)$ creates a *full marking*, by recursively marking the start states of m and all their descendants:

$$m_F(m) = m \cup \bigcup \{m_F(\Theta(s)) \mid s \in m, s \text{ is composite}\}$$

For example, applying m_F to automata **always** A, returns the set consisting of **always** A, s_{new}, A and any start states contained within A.

$Step$ transitions an automaton A from one full marking to the next:

$Step(A, {m_i}^{(t)}) \rightarrow {m_j}^{(t+1)}$ ::

1. $M1 := \{s \in {m_i}^{(t)} \mid s \text{ is primitive}\}$
2. $C := \bigwedge_{s \in M1} \mathcal{C}_P(s)$
3. $M2 := \bigcup_{s \in M1} \overline{\mathcal{T}_P}(s, C)$
4. return $m_F(M2)$

[1]Execution "completes" when no marks remain, since the empty marking is a fixed point.

$Step$ involves identifying the marked primitive states (Step 1), collecting the constraints of these marked states into a constraint store (Step 2), identifying the transitions of marked states that are enabled by the store and the resulting states reached (Step 3), and, initializing any automata reached by this transition (Step 4). The result is a full marking for the next time step.

To transition in step 3, let $(l_i, s_i) \in \mathcal{T}_P(s)$ be any transition pair of a currently marked primitive state s. Then s_i is marked in the next instant if l_i is entailed by the current constraint store, C (computed in step 2). A label 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 $\not\models c \in l_i.C \not\models c$.[2]

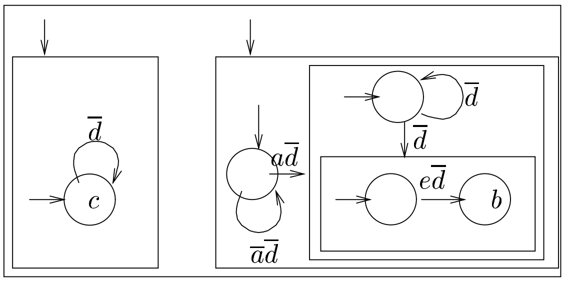
Applying $Step$ to the initial marking of **always** A causes s_{new} to transition to A and back to s_{new}, and for A to transition internally. The new mark on A invokes a second copy of A, by marking $A's$ start states. More generally, s_{new} is responsible for initiating A during every time step after the first. A transition back to itself ensures that s_{new} is always marked. The transition to A puts a new mark on 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 the compactness of this novel encoding, by avoiding the need for explicit copies of A.

7 A Simple Example

As an example consider the RMPL expression:

> **do**
> ($\textbf{always}$ c,
> **when** a **donext always if** e **thennext** b)
> **watching** d

This expression roughly maps to:



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

[2]Formally, $\overline{\mathcal{T}_P}(s, C) = \{s_i \mid (l_i, s_i) \in \mathcal{T}_P(s), C \models l_i\}$.

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.

8 Probabilistic HCA

We extend HCA to Markov processes by replacing the single initial marking and transition function of HCA with a probability distribution over possible initial markings and transition functions. We describe a probabilistic, hierarchical, constraint automata by a tuple $\langle \Sigma, \mathbf{P}_{\Theta}, \Pi, \mathcal{O}, \mathcal{C}_P, \mathbf{P}_{\mathcal{T}_P} \rangle$, where:

- Σ, Π, $\mathcal{O}$ and $\mathcal{C}_P$ are the same as for HCA.
- $\mathbf{P}_{\Theta}(m_i)$ denotes the probability that $m_i \subseteq \Sigma$ is the initial marking.
- $\mathbf{P}_{\mathcal{T}_P}(s_i)$, for each $s_i \in \Sigma_p$, denotes a distribution over possible transition functions $\mathcal{T}_P{}^j(s_i) : \mathcal{C}[\Pi] \to 2^{\Sigma}$.

The transition function $\mathbf{P}_{\mathcal{T}_P}(s_i)$ is encoded as an AND/OR tree. We present an example at the end of the next section, when describing the **choose** combinator.

PHCA execution is similar to HCA execution, except that m_F probabilistically selects an initial marking, and $Step$ probabilistically selects one of the transition functions in $\mathbf{P}_{\mathcal{T}_P}$ for each marked primitive state. The probability of a marking $m_i{}^{(t)}$ is computed by the standard belief update equations given in Section 2. This involves computing $\mathbf{P}_{\mathcal{T}}$ and $\mathbf{P}_{\mathcal{O}}$.

To calculate transition function $\mathbf{P}_{\mathcal{T}}$ for marking m_i recall that a transition $\mathcal{T}$ is composed of a set of primitive transitions, one for each marked primitive state s_i, and that the PHCA specifies the transition probability for each primitive state through $\mathbf{P}_{\mathcal{T}_P}(s_i)$. We make the key assumption that primitive transition probabilities are conditionally independent, given the current marking. This is analogous to the failure independence assumptions made by GDE[de Kleer and Williams, 1987] and Livingstone[Williams and Nayak, 1996], and is a reasonable assumption for most engineered systems. Hence, the composite transition probability between two markings is computed as the product of the transition probabilities from each primitive state in the first marking to a state in the second marking.

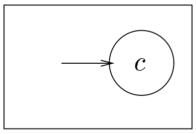
We calculate the observation function $\mathbf{P}_{\mathcal{O}}$ for marking m_i from the model, similar to GDE[de Kleer and Williams, 1987]. Given the constraint store C for m_i from step 2 of $Step$, we test if each observation in o_i is entailed or refuted, giving it probability 1 or 0, respectively. If no prediction is made, then an *a priori* distribution on observables is assumed (e.g., a uniform distribution of $1/n$ for n possible values).

This completes PHCA belief update. Our remaining tasks are to compile RMPL to PHCA, and to implement belief update efficiently.

9 Mapping RMPL to PHCA

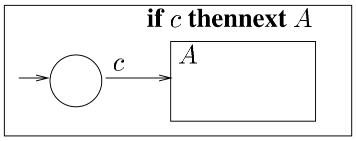
Each RMPL primitive maps to a PHCA as defined below. RMPL sub-expressions, denoted by upper case letters, are recursively mapped to equivalent PHCA.

c. Asserts constraint c at the initial instant of time:

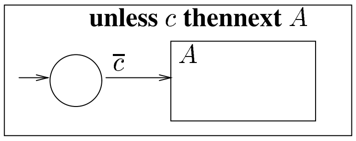


The start state has no exit transitions, so after this automaton asserts c in the first time instant it terminates.

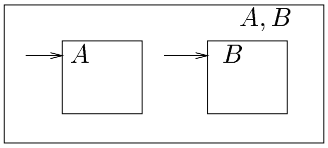
if c **thennext** A. Behaves like A in the next instant if the current theory entails c. Given the automaton for A, we add a new start state, and a transition from this state to A when c is entailed:



unless c **thennext** A. Executes A in the next instant if the current theory does *not* entail c. This mapping is analogous to **if** c **thennext** A. It is the only construct that introduces condition $\not\models c$. This introduces non-monotonicity; however, since these non-monotonic conditions hold only 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[Berry and Gonthier, 1992].

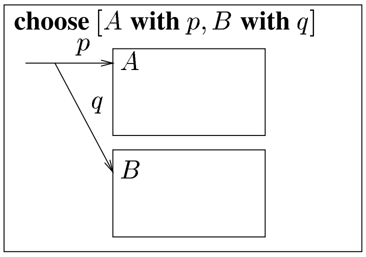


A, B. This is the parallel composition of two automata. The composite automaton has two start states, given by the two automata for A and B.

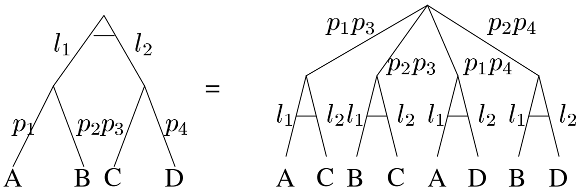


always A. Starts a new copy of A at each time instant, as described in Section 5.

choose $[A$ **with** p, B **with** $q]$. Reduces to A with probability p, to B with probability q, and so on. Recall that we required that all constraints asserted within A and B must be within the scope of a **next**. This ensures that 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. This is the only combinator that introduces probabilistic transitions.



Encoding probabilistic choice requires special attention due to the use of nested **choose** expressions. We encode the transition function $\mathcal{T}_P(s_i)$ as a probabilistic *AND-OR* tree (below, left), enabling a simple transformation from nested **choose** expressions to a PHCA.



In this tree each leaf is labeled with a set of one or more *target states* in Σ, which the automaton transitions to in the next time step. 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 $\mathbf{P}[b_{ij} \mid a_i]$. These are $p_1 \ldots p_4$ in the left tree. The probabilities of branches emanating from each OR node a_i sum to unity.

The branches of a deterministic *AND* node represent an inclusive set of choices. The node is indicated by a horizontal bar through its branches. Each branch is labeled by a set of conditions l_{ij}, as defined for HCA. These are l_1 and l_2 in the left tree. During a transition, every branch in an AND node is taken that has its label satisfied by the current state (*i.e.*, $\mathbf{P}[b_{ij} \mid a_i, l_{ij}] = 1$).

To map this tree to $\mathcal{T}_P(s_i)$, each AND-OR tree is compiled to a two level tree (shown above, right), with the root node being a probabilistic OR, and its children being deterministic ANDs. Compilation is performed using distributivity, shown by the figure, and commutativity. Commutativity 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 $\mathbf{P}(\mathcal{T}_P{}^j(s_i))$.

10 PHCA Estimation as Beam Search

We demonstrate PHCA belief update with a simple implementation of mode estimation, called RMPL-ME, that follows Livingstone[Williams and Nayak, 1996]. Livingstone tracks the most likely trajectories through the Trellis diagram by using beam search, which expands only the highest probability transitions at each step. To implement this we first modify *Step*, defined for HCA, to compute the likely states of $\sigma^{(\bullet t+1)}[m_i]$. This new version, Step_P, returns a set of markings, each with its own probability.

$Step_P(A, m_i{}^{(t)}) \rightarrow m_j{}^{(t+1)}$ ::

1. $M1 := \{s \in m_i{}^{(t)} \mid s \text{ is primitive}\}$
2. $C := \bigwedge_{s \in M1} \mathcal{C}_P(s)$

$3a.$ $M2a := \prod_{s \in M1} \overline{\mathcal{T}_P(s, C)}$

$3b.$ $M2b := \{(m_F(\bigcup_{i=1}^n \{s_i\}), \prod_{i=1}^n p_i) \mid \langle (s_1, p_1), \ldots, (s_n, p_n) \rangle \in M2a\}$

$3c.$ $m_j{}^{(t+1)} := \{(S, \sum_{(S,p) \in M2b} p) \mid (S, p) \in M2b\}$

4. return $m_j{}^{(t+1)}$

Step 3a builds the sets of possible primitive transitions. Step 3b computes for each set the combined next state marking and transition probability. Step 3c sums the probabilities of all composite transitions with the same target. Step 4 returns this approximate belief state. In Steps 3a and b, we enumerate transition sets in decreasing order of likelihood until most of the probability density space is covered (e.g., 95%). Best first enumeration is performed using our OPSAT system, generalized from [Williams and Nayak, 1996]. OPSAT finds the leading solutions to the problem "$arg \min f(\mathbf{x})$ subject to $C(\mathbf{x})$," where $\mathbf{x}$ is a state vector, $C(\mathbf{x})$ is a set of propositional state constraints, and $f(\mathbf{x})$ is an additive, multi-attribute utility function. OPSAT tests a leading candidate for consistency against $C(\mathbf{x})$. If it proves inconsistent, OPSAT summarizes the inconsistency (called a *conflict*) and uses the summary to jump over *leading* candidates that are similarly inconsistent.

After computing the leading states of $\sigma^{(\bullet t+1)}[m_i]$, RMPL-ME computes $\mathbf{P}_{\mathcal{O}}[m_i{}^{(t)} \mapsto o_i{}^{(t)}]$ using the constraint store extracted in step 2, and uses these results to compute the final result $\sigma^{(t+1\bullet)}[m_i]$, from the standard equation.

11 Implementation and Discussion

Implementations of the RMPL compiler, RMPL-ME and OPSAT are written in Common Lisp. The full RMPL language is an object-oriented language, in the style of Java, that supports all primitive combinators (Section 4) and a variety of defined combinators. The RMPL compiler outputs PHCA as its object code. RMPL-ME uses the compiled PHCAs to perform online incremental belief update, as outlined above. To support real-time embedded applications, RMPL-ME and OPSAT are being rewritten in C and C++.

The DS1 OpNav example provides a simple demonstration of RMP-ME. In addition RMPL-ME is being developed in two mission contexts. First, the C prototype is being demonstrated on the MIT Spheres formation flying testbed, a "robotic network" of three, soccer ball sized spacecraft that have flown on the KC-135. RMPL models are also being developed for the John Hopkins APL NEAR (Near Earth Asteroid Rendezous) mission.

Beam search is among the simplest of estimation approaches. It avoids an exponential blow up in the space of trajectories explored and avoids explicitly generating the Trellis diagram, but sacrifices completeness. Consequently it will miss a diagnosis if the beginning of its trajectory is sufficiently unlikely that it is clipped by beam search. A range of solutions to this problem exist, including an approach, due to [Hamscher and Davis, 1984], that uses a temporal constraint graph analogous to planning graphs. This encoding coupled with state abstraction methods has recently been incorporated into Livingstone [Kurien and Nayak, 2000], with attractive performance results. Another area of research is the incorporation of metric time. [Largouet and Cordier, 2000] introduces an intriguing approach based on model-checking algorithms for timed automata. Finally, [Malik and Struss, 1997] explores the discriminatory power of transitions vs state constraints in a consistency-based framework.

Acknowledgments

We would like to thank Michael Hofbaur, Howard Shrobe, Randall Davis and the anonymous reviewers for their invaluable comments. This research is supported in part by the DARPA MOBIES program under contract F33615-00-C-1702 and by NASA CSOC contract G998747B17.

A RMPL Defined Operators

To express complex behaviors in RMPL, we use the six RMPL primitives to define a rich set of constructs common to embedded languages, such as recursion, conditional execution, next, sequence, iteration and preemption. This section includes a representative sample of RMPL's derived constructs, used to support the DS1 opnav example.

Recursion and procedure definitions. A recursive declaration is of the form $P :: A[P]$, where A may contain occurrences of procedure name P. We implement this declaration with **always if** p **then** $A[p/P]$. At each time tick the expression looks to see if p is asserted (corresponding to p being invoked), and if so starts A. This method allows us to do parameterless recursion. Recursion with parameters is only guaranteed to be compilable into finite state automata if the recursion parameters have finite domains.

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.

A; B. This is the sequential composition of A and B. It performs A until A is finished. Then it starts B. It can be written in terms of the preceding 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. [Fromherz *et al.*, 1997] for details). For efficiency, the RMPL compiler implements A; B directly, rather than translating to basic combinators.

do A **while** c**.** Executes A, but if c is not true in a state, then A is terminated immediately. This combinator can be derived from the preceding combinators as follows:

do d **while** $c = c \to d$
do (**if** d **thennext** A) **while** $c =$
  if $c \wedge d$ **thennext do** A **while** c
do (A, B) **while** $c =$ **do** A **while** c, **do** B **while** c
do (**always** A) **while** $c = a$, **always if** $(a \wedge c)$ **then**
  (**do** A **while** c, **next** a)
do (**unless** d **thennext** A) **while** $c =$
  if c **then unless** d **thennext** (**do** A **while** c)
do (**choose** [A**with**p, B**with**q]) **while** $c =$
  choose (**do** A **while** c **with**p, **do** B **while** c **with**q)

For efficiency, RMPL derives the automaton for **do** A **while** c from the automaton for A by adding the label $\models c$ to all transitions in A, and in addition, replacing all the propositional formulas ϕ in the states by $c \to \phi$. Thus if c is not entailed by constraints outside of A, no transition or constraint in this automaton will be enabled.

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.

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. and is defined as:

when c **donext** $A =$
always if $(a \wedge c)$ **thennext** A, **do always** a **watching** c

References

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

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

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

[Fromherz *et al.*, 1997] Markus Fromherz, Vineet Gupta, and Vijay Saraswat. cc – A generic framework for domain specific languges. In *POPL Workshop on Domain Specific Languages*, 1997.

[Guernic *et al.*, 1991] P. Le Guernic, M. Le Borgne, T. Gauthier, and C. Le Maire. Programming real time applications with *signal*. In *Proc IEEE* [1991], pages 1321–1336.

[Halbwachs *et al.*, 1991] N. Halbwachs, P. Caspi, and D. Pilaud. The synchronous programming language *lustre*. In *Proc IEEE* [1991], pages 1305–1320.

[Hamscher and Davis, 1984] W. Hamscher and R. Davis. Diagnosing circuits with state: An inherently underconstrained problem. In *Proc AAAI-84*, pages 142–147, 1984.

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

[Kurien and Nayak, 2000] J. Kurien and P. Nayak. Back to the future for consistency-based trajectory tracking. In *Proceedings of AAAI-00*, 2000.

[Largouet and Cordier, 2000] C. Largouet and M. Cordier. Timed automata model to improve the classification of a sequence of images. In *Proc ECAI-00*, 2000.

[Malik and Struss, 1997] A. Malik and P. Struss. Diagnosis of dynamic systems does not necessarily require simulation. In *Proceedings DX-97*, 1997.

[Saraswat *et al.*, 1996] V. Saraswat, R. Jagadeesan, and V. Gupta. Timed Default Concurrent Constraint Programming. *J Symb Comp*, 22(5-6):475–520, November/December 1996.

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