The Reactive Model-based Programming Language

Mitch Ingham, Robert Ragno, Andreas Wehowsky, Brian Williams
MIT Space Systems and Artificial Intelligence Laboratories
Date of Last Revision: 05/23/02

Engineers like to reason about embedded systems in terms of state evolutions. However, embedded programming languages, such as Esterel [berry92], Signal [guernic91], Lustre [halbwachs91] and Statecharts [harel87], interact with a physical plant by reading sensor variables and writing control variables. Constraint programming languages, such as the Timed Concurrent Constraint Language (TCC) [gupta96], replace the traditional notion of a "store" as a valuation of variables with the notion of a store as a set of constraints on program variables. These languages interact with the store by "telling" and "asking" constraints at consecutive time points. In both embedded and constraint programming, it is the programmer's responsibility to encode the relationship between intended state and the sensors and actuators. This involves reasoning through a set of system interactions under a range of possible failure situations. The complexity of the interactions and the number of possible scenarios makes this an error-prone process.

A model-based programming language leverages the benefits of both embedded programming and constraint programming, with the key difference that it interacts directly with the plant state. This is accomplished by allowing the programmer to *read* or *write* constraints on "hidden" state variables in the plant, i.e. states that are not directly observable or controllable. It is then the responsibility of the language's execution kernel to map between hidden states and the plant sensors and control variables. This mapping is performed automatically by employing a deductive controller that reasons from a common-sense plant model.

A model-based program is comprised of two components. The first is a *control program*, which uses standard programming constructs to codify specifications of desired system behavior. In addition, to execute the control program, the execution kernel needs a model of the system it must control. Hence the second component is a *plant model*, which includes models of the plant's nominal behavior and common failure modes. This model unifies constraints, concurrency and Markov processes. The concepts of model-based programming can also be applied to the problem of temporal planning. A *temporal planning model* can be specified as a control program, augmented with notions of temporal constraints on activities and non-deterministic choice between activities.

The purpose of this document is to provide a specification for the first-generation of model-based programming language, called the *Reactive Model-based Programming Language* (RMPL). 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's object language is loosely modeled after Java, with the exception that it supports multiple inheritance. RMPL programs compile to a compact underlying computational model, based on hierarchical constraint-based automata. The constraint system currently supported by RMPL is a propositional state logic, where propositions are assignments of state variables, to values within their domains. RMPL's addition of reactive combinators to a constraint-based modeling language provides a more expressive language, with more flexibility in its representation. It allows, for instance, the encoding of partially observable Markov decision processes. In this document we discuss the subsets of RMPL used to specify control programs and plant models. We also present the extensions to RMPL that enable representations of temporal planning models.

Table of Contents

Introduction to RMPL

Numerous highly-autonomous aerospace systems, such as NASA's Deep Space One (DS1) spacecraft [bernard99], the next generation of space telescopes [ingham01], and various Mars Rover prototypes [kurien98], are being deployed that leverage many of the fruits of Artificial Intelligence research in 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 myriad of languages employed for the numerous software tasks running on a spacecraft processor. These tasks include sequencing, system monitoring, system reconfiguration, planning, and low-level control.

As a solution to this problem, we introduce the Reactive Model-based Programming Language (RMPL), which combines probabilistic, constraint-based modeling with reactive programming constructs, and offers a simple semantics in terms of partially observable Markov decision processes (POMDPs). RMPL can express a rich set of mixed hardware and software behaviors, and thus will provide a foundation for developing a unified model-based framework for autonomous spacecraft control, providing integrated sequencing, monitoring and fault protection capabilities [williams01].

RMPL represents a significant evolution from the Model-based Programming Language (MPL) used in the Livingstone mode estimation and reconfiguration system flown as part of the Remote Agent (RA) on DS-1 [muscettola98]. RMPL is an object-oriented language that, like MPL, describes co-temporal interactions between subsystems (constraints), and the evolution of these interactions over time. Like MPL, it provides a language for specifying, at a commonsense level, the behavior of complex embedded systems that react to external and internal stimuli. Using RMPL, plant models can be described both in terms of their nominal behavior, and their behavior under failure.

In addition to allowing specification of plant models, RMPL provides a suite of reactive programming constructs, similar to those in Esterel [berry92], or the Executive Support Language (ESL) used to develop the RA Smart Executive [gat96]. These constructs can be used to write control programs, which are specifications of the desired system behavior, and which operate directly on the hidden plant states described in the plant model. A model-based program is executed by automatically generating a control sequence that moves the physical plant to the states specified by the control program. The mapping from these hidden states to plant sensors and control variables is performed automatically, by a deductive controller that operates on the plant model.

The remainder of this document provides a specification of the RMPL language, starting with an overview of the fundamentals: notation and syntax, variable declaration, constraint system and object definition. The document then introduces the various RMPL constructs used in specifying control programs, plant models and temporal plan models.

This is followed by some examples of how RMPL is used in each of these types of models. Finally, the document concludes with a brief overview of the RMPL compiler, which compiles RMPL code into the appropriate computational model for execution by the model-based executive/planner.

Fundamentals

Notation and Syntax

The current implementation of RMPL follows the notation of functional languages such as Lisp and Scheme. Expressions are enclosed in parentheses and are composed with other expressions to form more complex structures. The notation is generally 'prefix', meaning that the first element in a parenthesized expression is the operator and the other elements are its arguments. This allows for a clear and precise statement of complex constructs.

RMPL specifications are indicated in this document by red type. The notation and grammar used in these specifications is as follows:

- Expressions in CAPS correspond to RMPL keywords.
- Expressions in regular type correspond to a user-specified entry.
- Expressions in *italicized* type correspond to general RMPL expressions. Restrictions on the form of such expressions are provided following the specification.
- Square braces [ ] indicate optional expressions.
- A '+' symbol following an expression means that a list of one or more such expressions can be included.
- A '*' symbol following an expression means that a list of zero or more such expressions can be included.
- A list of items within braces { }, with each item separated by a vertical bar '|', is used to indicate that one of the items applies. A similar list within [ ] indicates that one of the items applies, optionally.

Variables and Domains

At the core of RMPL is the use of propositional state logic to describe and reason about the behavior of a system. In propositional state logic, each proposition is an assignment (variable = value), where the possible values are elements in a finite set. The variables correspond to system states (modes), controls, observables or other dependent attributes. The domain of a given variable is the set of values that can be assigned to the variable.

Defining Variable Domains with DEFDOMAIN

Before a variable is to be used, its domain must be specified. In RMPL, domains consist of finite lists of discrete possible values; each value is a distinct symbol. Domain declarations are provided at the top level of an RMPL file, prior to use of the domain type in a variable declaration (see 'Defining Objects with DEFCLASS', below).

(DEFDOMAIN domain-name (member^{+}))
defines an enumerated type, domain-name, with a finite domain consisting of one or more members.

Examples:

`(DEFDOMAIN valve-state (open closed stuck-closed))`
defines a variable domain called **valve-state** with values *open*, *closed* and *stuck-closed*.

`(DEFDOMAIN valve-command (open-cmd close-cmd no-cmd))`
defines a variable domain called **valve-command** with values *open-cmd*, *close-cmd* and *no-cmd*.

`(DEFDOMAIN flow-level (zero low high))`
defines a variable domain called **flow-level** with values *zero*, *low* and *high*. Each of these values obviously corresponds to a range of continuous flow units.

Constraints: Propositional State Logic

As mentioned above, RMPL uses propositional state logic as its basis for knowledge representation. A proposition (variable = value) is, in general, a statement that has a truth assignment of TRUE or FALSE. Propositions are also sometimes referred to as **(positive) literals**. In general, RMPL constructs are conditioned on the current state of the physical plant, and they act on the plant state in the next time instant.

In developing an RMPL plant model of a physical system, the objective is to specify all the **constraints** on system states and attributes which hold at any given instant of time, and the constraints that describe how the state of the system can change from one time step to the next. Plant models expressed in RMPL use *tell* operations to assert constraints on plant variables, representing the known behavior in each mode.

In specifying an RMPL control program, the objective is to specify the desired behavior of the plant, by stating constraints that, when made true, will cause the plant to follow a desired state trajectory. State assertions are specified as constraints on plant state variables that should be made true. RMPL's model of interaction is in contrast to Esterel and TCC, which both interact with the program memory, sensors and control variables, but not with the plant state. Esterel interacts by emitting and detecting signals, while TCC interacts by *telling* and *asking* constraints on program variables. In contrast, RMPL control programs *ask* constraints on plant state variables, and request that specified constraints on state variables be *achieved* (as opposed to *tell*, which asserts that a constraint **is** true). State assertions in RMPL control programs are treated as *achieve* operations, while state condition checks are *ask* operations. It should be noted that, although RMPL provides the flexibility of asserting any form of constraint on plant

variables, the current implementation of the model-based executive, called *Titan*, can only handle assertions of constraints that are conjunctions of state variable assignments.

In this section, we outline the various constructs available to the RMPL modeler, which can be used to specify constraints. Within RMPL, constraints are expressed as well-formed formulae (WFFs). Though a variety of constructs are available for the purpose of facilitating the modeling task, the corresponding WFFs reduce to a set of one or more propositions, connected using the standard logical connectives (AND, OR, and NOT).

Basic Propositions

The following propositions are used as the basic building blocks for WFFs. They can be used within constraint assertions or condition checks, as indicated below.

- TRUE
- FALSE
  The most basic propositions, used as trivial constraints within RMPL conditions. TRUE is always entailed, whereas FALSE is never entailed.
- (variable = value)
  Basic variable assignment proposition. When used within an assertion in a control program, this specifies that *variable* should be made to have the specified *value*. When used within a condition check (in either a control program or a plant model), an assignment is determined to be entailed or not based on knowledge of the plant state.
- ( )
- (NOP)
- (NOTHING)
  An empty operation, used in control programs to assert no constraint (i.e. do nothing).

Primitive Logical Connectives

Each of these expressions is itself a WFF.

- (NOT *proposition*)
  Negation of the well-formed formula *proposition*.
- (OR *proposition*$^{+}$)
  Logical disjunction of one or more *proposition*s.
- (AND *proposition*$^{+}$)
  Logical conjunction of one or more *proposition*s.

Derived Logical Connectives

- (IMPLIES *antecedent consequent*)
  Creates an expression such that the WFF *antecedent* implies the WFF *consequent*. Corresponds to the equivalent WFF: (OR (NOT *antecedent*) *consequent*).

- (IF-THEN *condition true-wff*)
  Creates an expression such that entailment of the WFF *condition* implies *true-wff*.
- (IF-THEN-ELSE *condition true-wff false-wff*)
  Creates an expression such that entailment of the WFF *condition* implies *true-wff* and its non-entailment implies *false-wff*.
- (IFF *left-wff right-wff*)
  Creates an expression such that *left-wff* implies *right-wff* and *right-wff* implies *left-wff*.

Constraining Two Variables to be Equal

When modeling a physical system, it is often necessary to equate variables (i.e. to constrain their values to be equal). For example, when assembling component models into a subsystem model, it may be necessary to hook up one component's outputs to another component's inputs. In order to do this in RMPL, the "=" operator is used. It should be noted that the "=" operator is overloaded, as it is also used as the basic variable assignment operator. However, these uses are similar; both are statements that a variable takes on a particular value.

- (variable1 = variable2)
  Declares that two variables have the same value.

Example:

> The input and output flows of a valve might have the same value whenever the valve is open. If the flow can have the values ‘high’ or ‘low’, then the equivalence `(= flow-input flow-output)` translates to:
>
> ```
> (or
>     (and (= flow-output high)
>          (= flow-input  high))
>     (and (= flow-output low)
>          (= flow-input  low))
> )
> ```

Objects

RMPL is an object-oriented language. It provides a tool for system modeling based on the notions of objects and classes. A ‘class’ in RMPL corresponds to the type of object to be instantiated. Each class type has two kinds of members: *fields* and *methods*.

For a plant model specification, each class corresponds to a model for a physical component type (e.g. valve, driver, and camera components) or subsystem. Within a control program specification, each class corresponds to a set of control programs associated with a particular subsystem, sharing a common set of plant variables (e.g. a

class called ‘comm_subsystem’ might include control programs for ‘uplink’ and ‘downlink’ operations, each provided as a separate method). Finally, for a temporal planning model specification, a class corresponds to a set of macro activities in a plan domain, sharing a common set of variables.

Defining Objects with DEFCLASS

(DEFCLASS class-name (superclass-name) *member**)

Defines a structured type *class-name*, with associated members (fields and methods). *superclass-name* corresponds to a defined class, from which *class-name* inherits all members. The null superclass, denoted by “( )”, may be used if *class-name* does not inherit from any other class. Inheritance is monotonic, that is, there is no defaulting.

Object Members

A ***member*** is a field or method definition:

member ::= {*field-description* | *method-description*}
field-description ::= ({PUBLIC | PRIVATE} [STATE | CONTROL | OBSERVABLE | DEPENDENT] *class-name field-name*)
method-description ::= (*method-name* (*arglist*) *body*)

Fields and methods are described in more detail below.

Referring to Object Members

We refer to object members with *instance.member* [(*argument**)], where *member* is a field or method of the instantiated class *instance*, and the optional *arguments* correspond to parameters associated with a method call.

instance.member [(*argument**)]

Accesses a method or field relative to a class instance. In the case of a method, this corresponds to an invocation of the method. In the case of a field, this is a reference to the variable associated with the field. References can be made to members in the current instance (in which case the instance name need not precede the member name) or to members from another RMPL object (as permitted by the public/private scoping).

Fields

field-description ::= ({PUBLIC | PRIVATE} [STATE | CONTROL | OBSERVABLE | DEPENDENT] *class-name field-name*)

Denotes a field called *field-name* whose value is restricted to *class-name*, where *class-name* may denote an enumerated type (i.e. domain defined using DEFDOMAIN) or a structured type (i.e. another class defined using DEFCLASS). Access is restricted by the keywords PUBLIC and PRIVATE (although the current implementation does not enforce hiding the PRIVATE elements). PRIVATE fields

are visible only within an instance of the class; PUBLIC fields can be accessed by other class instances. The keywords STATE, CONTROL, OBSERVABLE and DEPENDENT can be used to specify a variable type, when *class-name* refers to an enumerated type:

- STATE denotes the field to be a state variable. Within an RMPL plant model, at most one STATE variable may be defined per class.
- CONTROL denotes control variables. Within RMPL plant models, these are associated with commands sent to the plant. Within RMPL control program specifications, these are associated with "local" variables, not associated with plant variables (e.g. counter variables, execution status variables, etc…).
- OBSERVABLE denotes variables associated with monitored system parameters (e.g. sensor measurements). Observable variables can be referenced in both plant models and control programs.
- DEPENDENT denotes variables associated with other (internal) attributes of the plant. Dependent variables are only specified for plant models.

For example, a plant model class called 'valve' might include the following field definitions:

`(public state valve-state mode)`

This introduces a field called *mode*, which is a state variable that ranges over an enumerated type *valve-state*. *mode* is accessible by methods outside of the scope of the 'valve' object.

`(public control valve-cmd command)`

This introduces a field called *command*, which is a control variable that ranges over an enumerated type *valve-cmd* (this means that this variable is hooked into the actual command being sent to the valve). *command* is accessible by methods outside of the scope of the 'valve' object.

`(public observable flow-type inflow)`

This introduces a field called *inflow*, which is an observable variable that ranges over an enumerated type *flow-type* (this means that some sensor measurement of the flow into the valve is available). *inflow* is accessible by methods outside of the scope of the 'valve' object.

Methods

method-description ::= (*method-name* (*arglist*) *body*)

Denotes a method called *method-name*, with parameters specified in *arglist*, and body code specified in *body*. In a plant model, methods are used to specify modal behavior and transitions for a modeled component of the plant. In a control program specification, methods correspond to control programs associated with a particular subsystem, sharing a common set of plant variables specified in the ***fields*** described

above. In a temporal planning model specification, methods correspond to macro activities that are available for insertion into a plan. All methods have public scope.

arglist and *body* are defined as follows:

arglist ::= parameter*

body ::= *rmpl_expression*

Parameter handling has been added to the RMPL compiler for use in temporal planning models, but it is not currently implemented for control programs and plant models. Currently, the RMPL compiler does not support type declaration/checking for parameters, although support for typed argument lists is under consideration.

In a control program specification, the *rmpl_expression* used in *body* is composed of any combination of RMPL constructs (presented below in 'Reactive Combinators') and RMPL method invocations of the form (*instance.method-name* (*arglist*)), as described above in 'Referring to Object Members'.

In a plant model specification, the *rmpl_expression* must include constraints on the component's allowed behavior in each mode and constraints on the allowed transitions between modes (see 'Plant Model Specification: Examples', below). This *rmpl_expression* must be in a form that allows it to be compiled to a concurrent transition system, as defined in [williams97].

In a temporal planning model specification, the *rmpl_expression* can be either an RMPL construct, a method invocation (called a "macro activity"), or a primitive activity invocation of the form (*activity-name* (*arglist*)), annotated with a time bound as discussed in 'Temporal Planning Constructs', below. A primitive activity invocation is distinguished from a macro activity invocation by virtue of the fact that macro activities are defined as methods in a class, whereas primitive activity invocations are not defined in any class within the temporal planning model. The plan runner dispatches primitive activity invocations in a plan to the executive. (Note: in the case of a model-based executive, these primitive activity invocations would correspond to method invocations in a control program.)

Root Specification

(root *root-class* [*root-name*])

A root can be specified for an RMPL program. This will be instantiated as the "top-level" object; only fields of this class (and subfields thereof) will be instantiated. If desired, a name can be given to the root object. Otherwise, the default *root-name* is taken to be the same as the *root-class*. ROOT provides the primary mechanism for object instantiation in RMPL.

Examples

The following control program excerpt illustrates how DEFCLASS is used to define a set of control programs. It also shows how ROOT is used to instantiate a class.

```
(defclass switch-control ()
  ;; FIELD
  (public state switch-state switch-mode) ;; assume 'switch-state' is a
valid domain, including values 'off' and 'on'

  ;; METHODS
  (turn-on ()
      (switch-mode = on)
  )
  (turn-off ()
      (switch-mode = off)
  )
)

(root switch-control switch-controller1)
```

The following plant model excerpt illustrates how DEFCLASS is used to define two instances of a simple plant component, and to combine them into a simple "subsystem". Note that these components are not realistic, in that they do not model potential stochastic fault behavior, only deterministic nominal behavior. ROOT is used to instantiate the simple subsystem, resulting in the creation of one instance of type two-switch-module, called *two-switches*, and two instances of type switch, *two-switches.switch1* and *two-switches.switch2*.

```
(defclass switch ()
  ;; FIELDS
  (public state switch-state mode) ;; assume 'switch-state' is a valid
domain, including values 'off' and 'on'
  (public observable on-off indicator-lamp) ;; assume 'on-off' is a
valid domain, including values 'off' and 'on'
  (public control on-off-cmd cmd) ;; assume 'on-off-cmd' is a valid
domain, including values 'turn-off' and 'turn-on'

  ;; METHOD
  (switch ()
     (always
        (parallel
        ;; Mode Constraints
           (if-then (mode = on) (indicator-lamp = on))
           (if-then (mode = off) (indicator-lamp = off))
        ;; Transitions
           (if-then (mode = on)
               (if-thennext-elsenext (cmd = turn-off) (mode = off)
(mode = on))
           )
           (if-then (mode = off)
```

```
               (if-thennext-elsenext (cmd = turn-on) (mode = on) (mode
= off))
           )
         )
      )
  )
)

(defclass two-switch-module ()
  ;; FIELDS
  (public switch switch1)
  (public switch switch2)
  ;; this class has no METHODS
)

(root two-switch-module two-switches)
```

Reactive Combinators

As mentioned above, RMPL expressions generally include some combination of reactive combinators. These combinators are fully orthogonal; that is, they may be nested and combined arbitrarily. RMPL constructs are closely related to constructs from the TCC programming language [gupta96] and Esterel [berry92]. In general, RMPL constructs are conditioned on the current state of the physical plant, and they act on the plant state in the next time instant.

RMPL defines the following reactive combinators:

g [MAINTAINING *m*]

Asserts that the plant should progress towards a state that achieves constraint *g*, while maintaining constraint *m* throughout. If *m* does not hold at any point, then assertion of *g* terminates immediately. The current model-based executive implementation cannot assert general constraints on state variables – only conjunctions of state variable assignments can be asserted. "MAINTAINING *m*" is optional (defaults to TRUE). This construct is the basic construct for making state assertions in RMPL control programs.

constraint

constraint is expressed as a WFF in propositional state logic. The simplest constraint expression is an assignment to a plant variable, such as (cmd = turn-on). 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. A constraint is 'entailed' if it is implied by the current state knowledge. Entailment is denoted by simply stating the constraint (e.g. c), non-entailment is denoted by using an overbar (e.g. $\bar{c}$). When used to assert a state constraint in a control program, this corresponds to the above construct *g*.

(IF-THENNEXT *condition exp*)

Conditional transition. If constraint *condition* is entailed in the current time step, then expression *exp* starts executing in the next time step.

(IF-THENNEXT-ELSENEXT *condition then-exp else-exp*)

Branch. If constraint *condition* is entailed in the current time step, then expression *then-exp* starts executing in the next time step. Otherwise (if *condition* is not entailed), starts executing expression *else-exp* in the next time step.

(UNLESS-THENNEXT *condition exp*)

Guarded transition. Unless constraint *condition* is entailed in the current time step, expression *exp* starts executing in the next time step. We also allow generalized sequences using IF-THEN and UNLESS-THEN, which we terminate by an IF-

THENNEXT or an UNLESS-THENNEXT. For example, (IF-THEN $cond_1$ (UNLESS-THENNEXT $cond_2$ exp)), which starts executing exp in the next instant if $cond_1$ is entailed and $cond_2$ is not entailed in the current time step.

(PARALLEL exp_1 exp_2 …)

Concurrency. Execution of each expression exp_i is initiated concurrently, starting in the current instant.

(SEQUENCE exp_1 exp_2 …)

Sequential composition. Starting with $i = 1$, 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, starting in the current time step.

(WHEN-DONEXT $condition$ exp)

Temporally-extended conditional transition. Wait until constraint $condition$ is entailed, then execute expression exp in the next time step.

(WHENEVER-DONEXT $condition$ exp)

Iterated conditional transition. Wait until constraint $condition$ is entailed, then execute expression exp in the next time step. This trigger can repeat indefinitely.

(DO-WATCHING $condition$ exp)

Preemption. Executes expression exp until completion or until constraint $condition$ is entailed. If $condition$ becomes entailed at any instant, it terminates execution of exp.

The above-mentioned RMPL constructs are used to encode control programs and deterministic plant models. This subset is sufficient to implement most of the control constructs of the Esterel language [ingham01b]. Note that RMPL can also be used to encode the probabilistic transition models capturing the behavior of the plant components, and to capture temporal constraints in planning models. The additional constructs required to encode such models are defined in the following two sections.

Constructs for Probability and Reward

In order to provide the full representational capability for POMDPs, the reactive combinators presented in the previous section must be augmented with constructs that capture stochastic effects and reward (or cost). For example, in a typical plant model, transitions between modes are probabilistic, representing the possibility of faulty or uncertain behavior. In this section we introduce two new constructs, representing probabilistic choice and choice between behaviors with different rewards/costs.

(CHOOSE-PROBABILITY *case*$^+$)
where *case*$_i$ ::= (*probability*$_i$ *exp*$_i$)
Markov Processes. System transitions to exactly one case, *case*$_i$, where it executes its expression *exp*$_i$. System chooses *exp*$_i$ with probability *probability*$_i$. The sum over *i* of *probability*$_i$ should equal 1. To preserve causality, we would like to ensure that the current set of constraints do not depend upon probabilistic choices made in the current state. We achieve this by restricting all constraints asserted within *exp*$_i$ to be within the scope of an IF-THENNEXT or UNLESS-THENNEXT.

(CHOOSE-REWARD *case*$^+$)
where *case*$_i$:= (*reward*$_i$ *exp*$_i$)
Decision Processes. System chooses a transition to exactly one case, *case*$_i$, associated with expression *exp*$_i$. System receives reward *reward*$_i$ for choosing *case*$_i$.

Temporal Planning Constructs

In order to be able to express temporal planning models in RMPL, the set of reactive combinators must be augmented with constructs representing time bounds and non-deterministic branching.

([*lower-bound*, *upper-bound*] *exp*)
(*exp* [*lower-bound*, *upper-bound*])
(*label* [*lower-bound*, *upper-bound*] *exp*)
Time bound placed on an expression. If a time bound is specified, a label may also be provided (at any position – order is unimportant). This label is ignored, but it will be used in generating the symbols in the output. If only a label is desired, an empty time bound of [] can be specified.

(CHOOSE exp^{+})
Non-deterministic branch point. System non-deterministically chooses a transition to exactly one expression exp_i.

This completes the introduction of the RMPL language. In the following three sections of this document, we provide examples of how RMPL is used to specify control programs, plant models and temporal planning models.

Control Program Specification: Examples

In this section, we provide two examples of RMPL control programs.

Example 1- Switch Controller

In this example, we specify a class called ‘switch-control’, which defines a set of very simple control programs (i.e. methods) that operate on the state of a switch component. These control programs serve to demonstrate the use of some basic reactive combinators. We then specify a class called ‘two-switch-control’, which defines a set of control programs that reference the controllers for two switch components. These control programs introduce some more complex behaviors, including method invocation, state maintenance, and “catch/throw” mechanisms. Finally, this example instantiates a two-switch-control object called ‘two-switches’, using the ROOT construct.

```
(defdomain switch-state (on off broken))

(defclass switch-control ()
  ;; FIELDS
  (public state switch-state switch-mode)

  ;; METHODS
  (turn-on ()
      (switch-mode = on)
  )
  (turn-off ()
      (switch-mode = off)
  )
  (turn-on-only-if-off ()
      (if-thennext (switch-mode = off)
                   (switch-mode = on))
  )
  (turn-off-unless-already-off ()
      (unless-thennext (switch-mode = off)
                       (switch-mode = off))
  )
  (keep-off ()
      (always (switch-mode = off))
  )
)

(defclass two-switch-control ()
  ;; FIELDS
  (public switch-control switch1-cmds)
  (public switch-control switch2-cmds)

  ;;METHODS
  (turn-one-on-two-off ()
      (parallel (switch1-cmds.turn-on())
                (switch2-cmds.turn-off())))
  )
  ;; we want one of the two indicator lamps to be on (one and only one!)
  (turn-on-one-not-both ()
      (if-thennext-elsenext (switch1-cmds.switch-mode = on)
```

```
                          (switch2-cmds.turn-off())
                          (switch2-cmds.turn-on()))
  )
  ;; in terms of primitive combinators:
  (turn-on-one-not-both2 ()
      (parallel (if-thennext (switch1-cmds.switch-mode = on)
                             (switch2-cmds.switch-mode = off))
                (unless-thennext (switch1-cmds.switch-mode = on)
                                 (switch1-cmds.switch-mode = on))
      )
  )
  ;; Now we want to ensure that this state is maintained, or else bail out.
  (maintain-one-on-not-both-with-error-handling ()
      (sequence (parallel (if-thennext (switch1-cmds.switch-mode = on)
                                       (switch2-cmds.switch-mode = off))
                          (unless-thennext (switch1-cmds.switch-mode = on)
                                           (switch1-cmds.switch-mode = on))
                )

                (do-watching (OR (AND (switch1-cmds.switch-mode = off)
                                      (switch2-cmds.switch-mode = off))
                                 (AND (switch1-cmds.switch-mode = on)
                                      (switch2-cmds.switch-mode = on)))
                             (nothing)
                )
      )
  )
)

(root two-switch-control two-switches)
```

Example 2- Orbital Insertion

The following example demonstrates how RMPL can be used to express more "realistic" behaviors, relevant to spacecraft systems. It also shows how model-based programming enables the programmer to focus on specifying the desired state evolutions of the system. Consider the task of inserting a spacecraft into orbit around a planet, such as Jupiter or Saturn. Our spacecraft includes a science camera and two identical redundant engines (Engine A and B), shown in Figure 1. An engineer thinks about this maneuver in terms of state trajectories:

Heat up both engines (called "standby mode"). Meanwhile, turn the camera off, in order to avoid plume contamination. When both are accomplished, thrust one of the two engines, using the other engine as backup in case of primary engine failure.

This specification is far simpler than a control program that must turn on heaters and valve drivers, open valves and interpret sensor readings for the engine shown in the figure. Thinking in terms of more abstract hidden states makes the task of writing the control program much easier, and avoids the error-prone process of reasoning through low-level system interactions. In addition, it gives the program's execution kernel the latitude to respond to novel failures as they arise. This is essential for achieving high levels of robustness.

The RMPL control program, shown below, codifies the above informal specification. State assertions are highlighted in blue type, while state condition checks are highlighted in red type. Recall that, to perform orbital insertion, one of the two engines must be fired. We start by concurrently placing the two engines in standby and by shutting off the camera. This is performed by lines 4-6, where these assertions are made in the context of a PARALLEL construct. We then fire an engine, choosing to use Engine A as the primary engine (lines 7-9) and Engine B as a backup, in the event that Engine A fails to fire correctly (lines 10-11). Engine A starts trying to fire as soon as it achieves standby and the camera is off (line 8), but aborts if at any time Engine A is found to be in a failure state (line 7). Engine B starts trying to fire only if Engine A has failed, B is in standby and the camera is off (line 10).

The orbital insertion example highlights five design features of RMPL. First, the program exploits full concurrency, by specifying parallel threads of execution; for example, the camera is turned off and the engines are set to standby in parallel (lines 3-12). Second, it involves conditional execution; for example, the control program must check for two conditions, prior to firing Engine A: that the engine to be fired is in standby mode, and that the camera is turned off (line 8). Third, it involves iteration; for example, line 8 says to iteratively test until Engine A is in standby and the Camera is off, and then to proceed. Fourth, the program involves preemption; for example, if the primary engine fails, the act of firing it should be preempted, in favor of firing the backup engine (lines 7-9). These four features are common to most synchronous reactive programming languages. As highlighted previously, the fifth and defining feature of RMPL is the ability to reference

hidden states of the physical plant within assertions and condition checks, such as (WHEN-DONEXT (AND (EngineA = Standby) (Camera = Off)) (EngineA = Firing), on lines 8-9.

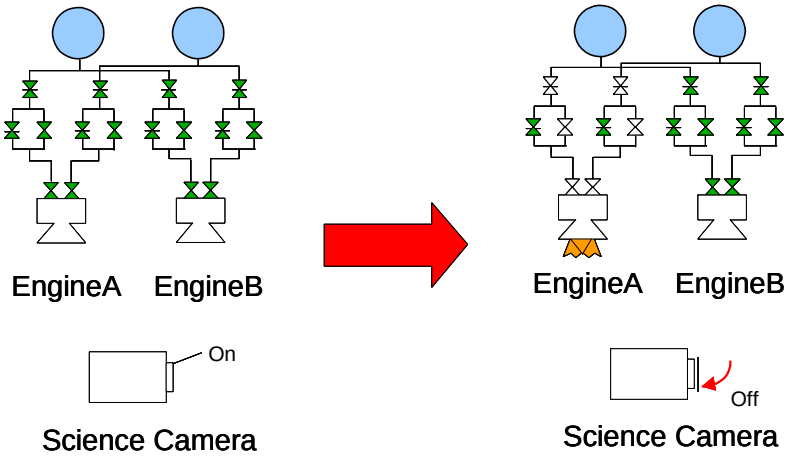

Figure 1. Orbital Insertion for Simplified Spacecraft

```
(defdomain engine-state (Off Standby Firing Failed))
(defdomain camera-state (Off On Failed))

(defclass spacecraft ()
  ;; FIELDS
  (public state engine-state EngineA)
  (public state engine-state EngineB)
  (public state camera-state Camera)

  ;; METHODS
1 (OrbitInsert ()
2    (do-watching (OR (EngineA = Firing) (EngineB = Firing))
3       (parallel
4          (EngineA = Standby)
5          (EngineB = Standby)
6          (Camera = Off)
7          (do-watching (EngineA = Failed)
8             (when-donext (AND (EngineA = Standby) (Camera = Off))
9                (EngineA = Firing)))
10         (when-donext (AND (EngineA = Failed) (EngineB = Standby) (Camera = Off))
11               (EngineB = Firing)))
12      )
13   )
14)

(root spacecraft sc1)
```

Plant Model Specification: Examples

In this section, we provide two examples of RMPL plant models.

Example 1- Camera component

RMPL can be used to model physical plants as a set of probabilistic concurrent transition systems, one for each component in the plant. The current implementation of the model-based executive operates on a specific form of concurrent transition system, called Concurrent Constraint Automata (CCA). Each state of an automaton, called a mode, has an associated constraint over the plant variables. These are used to represent co-temporal interactions. Transitions between modes are probabilistic, and are used to represent failures and uncertainty.

A graphical representation of a CCA model for a simple camera component is given in Figure 2. The camera model has one input, associated with the *power_in* variable, and one observable output, associated with the *shutter* variable. As depicted, there are three possible states for a camera object: *off*, *on*, and *failed*. The *off* and *on* states correspond to "nominal" modes of the camera, while the *failed* state is a "fault" mode. Each of the states is described by a set of constraints that must hold when that state is current (e.g. in the *off* mode, the value of the *power_in* variable must be 'zero' and the value of the *shutter* variable must be 'closed'). Transitions between these states are represented by arrows in the figure (green for nominal transitions, triggered by explicit commands; red for fault transitions, with some associated probability value).

As exemplified in the RMPL code below, to build a probabilistic constraint automaton for the camera component, we use the logical operator IF-THEN to associate constraints with modes (lines 4-7). To describe probabilistic transitions we use CHOOSE-PROBABILITY (line 8), containing nested IF-THENNEXT-ELSENEXT expressions to produce transitions between modes (lines 10-21, 22). Both forms of expressions are contained within an ALWAYS to enforce their behaviors for all time (line 2).

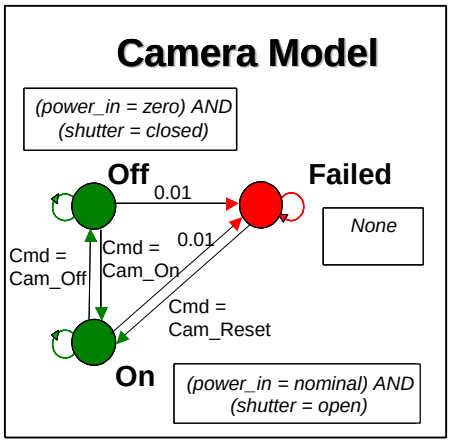

Figure 2. CCA Model for Camera Component

```
(defdomain camera-state (Off On Failed))
(defdomain power-setting (Zero Nominal))
(defdomain shutter-setting (Open Closed))
(defdomain camera-command (Cam_On Cam_Off Cam_Reset No-Cmd))

(defclass Camera ()
  ;; FIELDS
  (public observable shutter-setting shutter)
  (public dependent power-setting power_in)
  (public control camera-command Cmd)
  (public state camera-state Camera)

  ;; METHODS
1 (Camera ()2    (always
3       (parallel
4          (if-then (Camera = On)
5              (AND (power_in = nominal) (shutter = open)))
6          (if-then (Camera = Off)
7              (AND (power_in = zero) (shutter = closed)))
8          (choose-probability
9             (0.99 (parallel
10                   (if-then (Camera = On)
11                      (if-thennext-elsenext (Cmd = Cam_Off)
12                         (Camera = Off)
13                         (Camera = On)))
14                   (if-then (Camera = Off)
15                      (if-thennext-elsenext (Cmd = Cam_On)
16                         (Camera = On)
17                         (Camera = Off)))
18                   (if-then (Camera = Failed)
19                      (if-thennext-elsenext (Cmd = Cam_Reset)
20                         (Camera = On)
21                         (Camera = Failed))))
22             (0.01 (if-thennext TRUE (Camera = Failed)))
23           )
24       )
25    )
26)

(root camera Camera1)
```

Example 2: Valve and Driver - Mapping between MPL and RMPL

For Livingstone, the MPL modeling language was used to represent plant models. MPL models are compiled to CCA, for use by the Livingstone deductive engine. Here, through an illustrative example, we show how RMPL can also be used to represent plant models that compile to CCA, by presenting equivalent plant models expressed in both MPL and RMPL. This example also shows how multiple component classes are combined into a "module" class for instantiation.

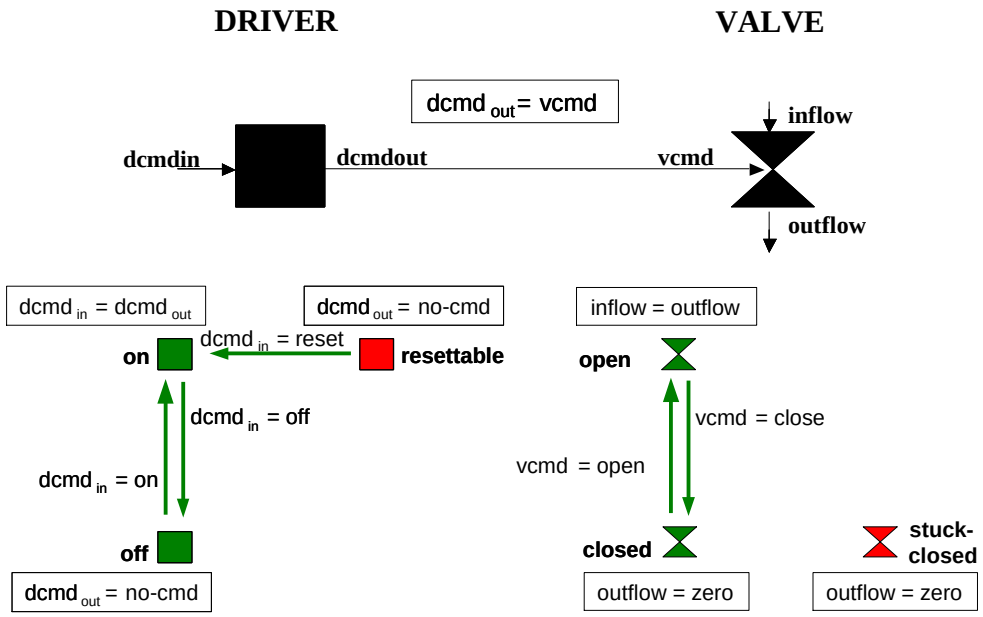

Figure 3. CCA Model for Driver-Valve Module

When using RMPL to create a plant model for a system (i.e. compiling to concurrent automata), DEFCLASS is used to define components or modules consisting of multiple components (corresponding to the DEFCOMPONENT and DEFMODULE constructs in MPL). The fields in each class correspond to the attributes defined within the :inputs, :outputs, and :attributes fields of the DEFCOMPONENT or DEFMODULE construct, as well as the :structure field in DEFMODULE. Consider the following RMPL and MPL models for the driver-valve module (Figure 3).

RMPL Version

```
(defdomain valve-state (open closed stuck-closed))
(defdomain driver-state (off on resettable))
(defdomain driver-cmd (driver_on driver_off driver_reset valve_open valve_close no-
cmd))
(defdomain flow-level (zero positive))

(defclass valve ()
    (public state valve-state mode)
    (public dependent driver-cmd cmd)
    (public dependent flow-level inflow)
    (public dependent flow-level outflow)
1  (valve ()
2      (always
3         (parallel
4            (if-then (mode = open)
5                (inflow = outflow))
6            (if-then (mode = closed)
7                (outflow = zero))
8            (if-then (mode = stuck-closed)
9                (outflow = zero))
10           (choose-probability
11               (0.999 (parallel
12                           (if-then (mode = open)
13                              (if-thennext-elsenext (cmd = valve_close)
14                                 (mode = closed)
15                                 (mode = open)))
16                           (if-then (mode = closed)
17                              (if-thennext-elsenext (cmd = valve_open)
18                                 (mode = open)
19                                 (mode = closed)))
20                           (if-thennext (mode = stuck-closed)
21                              (mode = stuck-closed))))
22                (0.001 (if-thennext TRUE (mode = stuck-closed)))
23             )
24        )
25   )

(defclass driver ()
    (public state driver-state mode)
    (public control driver-cmd cmd_in)
    (public dependent driver-cmd cmd_out)
1  (driver ()
2      (always
3         (parallel
4            (if-then (mode = on)
5                (cmd_in = cmd_out))
6            (if-then (mode = off)
7                (cmd_out = no-cmd))
8            (if-then (mode = resettable)
9                (cmd_out = no-cmd))
10           (choose-probability
11               (0.99 (parallel
12                           (if-then (mode = on)
13                              (if-thennext-elsenext (cmd_in = driver_off)
14                                 (mode = off)
15                                 (mode = on)))
16                           (if-then (mode = off)
17                              (if-thennext-elsenext (cmd_in = driver_on)
18                                 (mode = on)
19                                 (mode = off)))
20                           (if-then (mode = resettable)
```

```
21                                (if-thennext-elsenext (cmd_in = driver_reset)
22                                   (mode = on)
23                                   (mode = resettable)))
24              (0.01 (if-thennext TRUE (mode = resettable)))
25            )
26         )
27   )

(defclass valve-and-driver ()
  (public valve valve1) ;; instantiates a valve object called valve1
  (public driver driver1) ;; instantiates a driver object called driver1
  (valve-and-driver ()
     (always
         (driver1.cmd_out = valve1.cmd)
     )
  )
)

(root valve-and-driver VD1)
```

MPL Version

```
(defvalues driver-cmd (driver_on driver_off driver_reset valve_open valve_close no-
cmd))
(defvalues flow-level (zero positive))

(defcomponent valve (?name)
  (:inputs
   ((cmd ?name) :type driver-cmd :documentation "Command to valve")
   ((inflow ?name) :type flow-level :documentation "Flow into the valve"))
  (:outputs
   ((outflow ?name) :type flow-level :documentation "Flow out of the valve"))

 (:background
  :initial-mode closed)

 (open  :type :ok-mode
       :documentation ""
       :model (= (inflow ?name) (outflow ?name))
       :transitions ((close
                      :when (= (cmd ?name) valve-close)
                      :next closed)
                     (:otherwise :persist)))

 (closed  :type :ok-mode
         :documentation ""
         :model (and (= (inflow ?name) zero) (= (outflow ?name) zero))
         :transitions ((open
                        :when (= (cmd ?name) valve-open)
                        :next open)
                       (:otherwise :persist)))

 (stuck-closed :type :fault-mode
              :probability 0.001
              :documentation ""
              :model (and (= (inflow ?name) zero) (= (outflow ?name) zero))
              :transitions ((:otherwise :persist)))
)


(defcomponent driver (?name)
 (:inputs
  ((cmd_in ?name) :type driver-cmd :documentation "Command to driver"))
```

```
 (:outputs
  ((cmd_out ?name) :type driver-cmd :documentation "Output to driven device"))

 (:background
  :initial-mode off)

 (off  :type :ok-mode
       :documentation ""
       :model (= (cmd_out ?name) no-cmd )
       :transitions ((turn-on
                    :when (= (cmd_in ?name) driver-on)
                    :next on)
                   (:otherwise :persist)))

 (on  :type :ok-mode
      :documentation ""
      :model  (= (cmd_in ?name) (cmd_out ?name))
      :transitions ((turn-off
                   :when (= (cmd_in ?name) driver-off)
                   :next off)
                  (:otherwise :persist)))

 (resettable  :type :fault-mode
              :probability unlikely
              :model (= (cmd_out ?name) no-cmd )
              :documentation ""
              :transitions ((reset
                           :when (= (cmd_in ?name) driver-reset)
                           :next on)
                          (:otherwise :persist))))

(defmodule valve-and-driver (?name)
  (:structure
   (driver (driver1 ?name))
   (valve (valve1 ?name))
  )
  (:connections
   (= (cmd_out (driver1 ?name)) (cmd_in (valve1 ?name)))
  )
)

(defun create-valves()
  ;; Instantiate the model, monitors and commands within WITH-MRP,
  ;; so that the reactive planner will know about the model. We
  ;; can later ask the reactive planner for recovery plans.

 (with-mrp
   ;; create a driver-and-valves module, which will
   ;; instantiate all of its subcomponents
   (instantiate-module '(valve-and-driver VD1))

   (new-command-table)
   (new-monitor-table)

   (new-command '(cmd_in (driver1 VD1)))
 )
)
```

Temporal Planning Model Specification: Examples

In this section, we provide two examples of RMPL temporal planning models.

Example 1- Cooperative Rovers

This RMPL temporal planning example describes movements of three rovers. The example shows how to use sequential and parallel RMPL constructs with time bound annotations to describe concurrent activities for the rovers. The program does not use non-deterministic choice or symbolic constraints. In this case, the main job for the planner is simply to schedule the activities into a temporally consistent plan.

The rover class declared below has three movement macro activities: {*move1*, *move2*, *move360*}. *move1* and *move2* make a rover drive forward, backward and turn (see the details in the code). *move360* forces a rover to rotate in place at 90 degrees/sec in 4 seconds. The commands {*backward*, *forward*, *stop*, *turn_left*, *turn_right*} are all rover-specific primitive activity invocations that are executed directly on the hardware. Note that all time units are in milliseconds.

The program also shows how parameters can be passed to primitive invocations. Both numbers and strings are valid activity arguments. Note that parameters also can be passed to macro activities. The planner distinguishes between macro activities and primitive invocation by checking to see if an activity exists in the macro activity library (i.e. defined as a method in the planning model). If it does not, the activity is considered to be a primitive invocation, which is passed down to the executive.

```
(defclass rover ()
   (move1 ()
       (sequence
           ((forward(.1)) [1000,2000])         ;; move forward at 0.1 m/s
           ((backward(.1)) [1000,2000])        ;; move backward at 0.1 m/s
           ((stop())[0,1000])                  ;; stop rover
       )
   )

   (move2 ()
       (sequence
           (parallel
               ( (backward(.1) ) [1000,1500] )
               ( (turn_left(20) ) [1000,1500] ) ;; turn left at 20 degrees/sec
           )
           (parallel
               ( (forward(.1) ) [1000,1500] )
               ( (turn_left(.1) ) [1000,1500] )
           )
           (parallel
               ( (backward(.1) ) [1000,1500] )
               ( (turn_right(20)) [1000,1500] ) ;; turn right at 20 degrees/sec
           )
```

```
            ( (stop() ) [0,1000] )
        )
    )

    (move360()
        (sequence
            ((turn_left(90))[4000,4500]) ;; turn 90 degrees/sec in 4 seconds
            ((stop())[0,100])
        )
    )
)


(defclass rover_team ()
    (public state rover Foto)       ;; instantiate three rovers
    (public state rover Notorious)
    (public state rover Gaia)
    (dance()                        ;; dance is the main activity
        (sequence
            ( parallel
                ( (Foto.move1() ) [2000,5000] )          ;; rover1
                ( (Notorious.move1() ) [2000,5000] )     ;; rover2
                ( (Gaia.move360() ) [3000,5000] )        ;; rover3
            )
            ( parallel
                ( (Foto.move2()) [3000,6000] )
                ( (Notorious.move2()) [3000,6000] )
                ( (Gaia.move2()) [3000,6000] )
            )
        )
    )
)


(root rover_team) ;; root object of RMPL program
```

Example 2- Science and Downlink

The following Science and Downlink scenario is a temporal planning example that uses many of the RMPL keywords and both temporal and symbolic (i.e. state) constraints.

In this example, our spacecraft must plan to perform one of two possible observations, obsA or obsB, followed by a downlink activity. The initial spacecraft position is assumed to be posA (the position that enables obsA to be performed). So if obsB is chosen, the spacecraft must be reoriented. The planner needs to choose between these two "threads" of execution:
1) hold current position while performing obsA;
2) reorient spacecraft, then hold new position while performing obsB.

Following either observation is a downlink task, modeled in the *downlink* macro activity. Based on the current position of the spacecraft and the knowledge of which of its two omnidirectional antennae is currently active, the spacecraft is reoriented if necessary. Thereafter, we wait for a communication window to be open and then downlink the data that was recorded during the observation. When the RMPL planning model is compiled, the *downlink* macro activity is merged into the *obsdown* temporal plan network.

The main activity is defined in method *obsdown* in the *sc* class. The example references one macro activity, *downlink()*, and several primitive activities: *perform_obsA()*, *perform_obsB()*, *analyze_data()*, *packetize_data()*, *reorient_sc_to_A()*, and *reorient_sc_to_B()*. Note that all time units are in minutes.

```
(defdomain orientation (posA posB))             ;; orientation of s/c
(defdomain comm_window (window_ok no_window)    ;; domain of comm windows
(defdomain omni (omniA omniB))                  ;; active/desired antenna

(root sc) ;; root object of RMPL program

(defclass sc ()
    (public state orientation pos)
    (public state comm_window comm)
    (public state omni antenna)

    (obsdown() [1,1000] ;; real bounds are 125,162
        ;; Sequence of observations and data analysis
        ;; Assume initial s/c orientation is A.
        ;; Choose between observation A or B. B requires the spacecraft to
        ;; be reoriented.
        (choose
            (parallel ;; observation A
                (sequence
                    ((perform_obsA ()) [25,35]) ;; command passed to executive
                    ((analyze_data ()) [5,10])  ;; command passed to executive
                    ((packetize_data ()) [5,10]);; command passed to executive
                    ((downlink()) [90,100]) ;; call downlink macro activity
                )
                ((pos = posA) [125,155]) ;; assert that position is A
            )
```

```
            (sequence
                ((reorient_sc_to_B()) [5,7])
                (parallel ;; observation B
                    (sequence
                        ((perform_obsB ()) [25,35])
                        ((analyze_data ()) [5,10])
                        ((packetize_data ()) [5,10])
                        ((downlink()) [90,100]) ;; call downlink macro activity
                    )
                    ((pos = posB) [132,162]) ;; assert that position is now B
                )
            )
        )
    )

    (downlink ()
        (parallel
            ;; Communication window constraints
            (comm_win []
                (sequence
                    ((comm = NOT(window_ok)) [80,80])
                    ((comm = window_ok) [10,20])
                )
            )
            ;; The nominal case: we are in posA and omniA works
            ;; so a reorientation is not needed.
            (sp_antenna [] ;; modify if failure
                ((antenna = omniA) [0,100])
            )
            (orient_and_downlink []
                (sequence
                    (choose_orientation []
                        (choose
                            ;; two cases where reorientation is necessary
                            (if-thennext (AND (antenna = omniA) (pos = posB))
                                (move_to_A (reorient_sc_to_A()) [5,10])
                            )
                            (if-thennext (AND (antenna = omniB) (pos = posA))
                                (move_to_B (reorient_sc_to_B()) [5,10])
                            )
                            ;; in case reorientation is not necessary
                            (if-thennext (AND (antenna = omniA) (pos = posA))
                                (no_move_A (nop()) [0,0])
                            )
                            (if-thennext (AND (antenna = omniB) (pos = posB))
                                (no_move_B (nop()) [0,0])
                            )
                        )
                    )
                    ;; When data is ready and comm window is open, initiate downlink

                    ( when-donext (comm = window_ok)
                        (do-watching (NOT(comm = window_ok))
                            ((DownlinkDataBlock()) [3,5]) ;; cmd passed to executive
                        )
                    )
                )
            )
        )
    )
)
```

The RMPL Compiler

In this section, we briefly discuss the RMPL compiler, and its implementation for the Titan model-based executive and the Kirk temporal planner.

We address separately the compilation of control programs, the compilation of plant models and compilation of activity planning models.

Compiling RMPL Control Programs

The RMPL control program is executed by Titan's control sequencer. Executing a control program involves compiling it to a variant of hierarchical automata, called *hierarchical constraint automata* (HCA), and then executing the automata in coordination with the deductive controller. The formal specification of HCA is given in [williams02]. The compiler (written in C++) takes in an RMPL file containing a set of control programs (as illustrated in the previous examples), converts the code to HCA according to the mapping in Figure 4, and writes the resulting HCA to an output file, which can be read in and executed by Titan's control sequencer.

To illustrate the compilation of RMPL control programs into automata, consider the RMPL code for orbital insertion, presented earlier. The RMPL compiler converts this to the corresponding HCA, shown in Figure 5. Execution of the HCA by the model-based executive is discussed in [williams02].

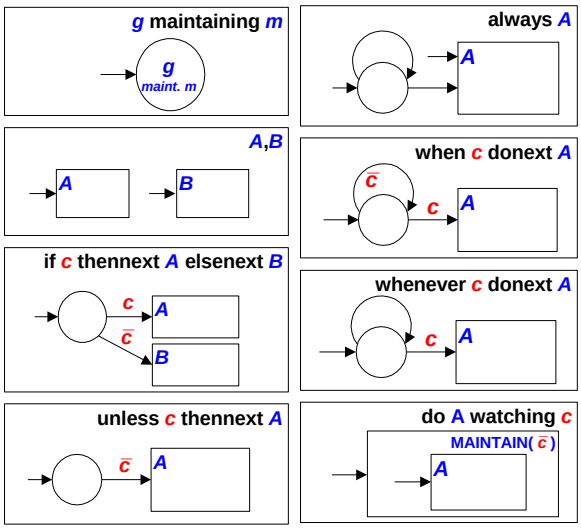

Figure 4. Mapping from RMPL Constructs to HCAs

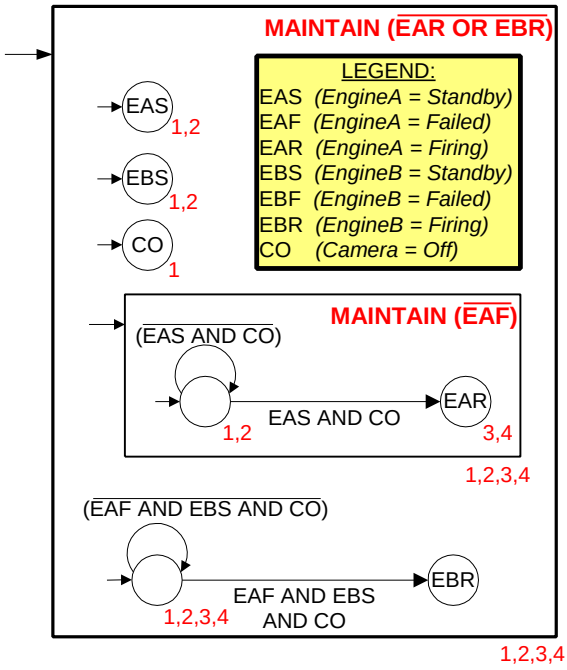

Figure 5. HCA Representation for the Orbital Insertion Control Program

Compiling RMPL Plant Models

The compiler that converts RMPL plant models to CCA has not yet been fully implemented. The current implementation of Titan uses plant models written in MPL, and borrows Livingstone's MPL compiler (written in Common Lisp) to generate the CCA file. Titan's deductive controller takes in the CCA file and compiles it into a propositional logic theory, which can be reasoned through, in order to perform mode estimation and reconfiguration.

Compiling RMPL Temporal Planning Models

The compiler used to compile RMPL control programs to HCAs is also used to compile temporal planning models. The compiler has been augmented to recognize RMPL planning time bounds and non-deterministic choice constructs. The RMPL compiler produces a timed HCA, which is a derivative of the HCA described above, designed to contain information about lower and upper bounds of activities and to model decisions. There is a direct mapping from timed HCA to Temporal Plan Networks (TPNs), which is the input format for the model-based planner. A timed HCA to TPN converter reads in the compiled RMPL program in HCA format and converts the hierarchical automaton

data structure to a graph (TPN). The fact that the compilation is a *two-step* process is irrelevant to the model-based planner software and to the RMPL programmer.

References

[berry92] G. Berry, G. Gonthier, The Esterel programming language: Design, semantics and implementation, Science of Computer Programming 19~(2) (1992) 87 -- 152.

[guernic91] P. Le Guernic, M. Le Borgne, T. Gauthier, C. Le Maire, Programming real time applications with Signal, in: Proceedings of IEEE, Vol. 79:9, pp. 1321--1336 (1991) 1321--1336.

[halbwachs91 N. Halbwachs, P. Caspi, D. Pilaud, The synchronous programming language Lustre, in: Proceedings of IEEE, Vol. 79:9, pp. 1305--1320 (1991) 1305--1320.

[harel87] D. Harel, Statecharts: A visual approach to complex systems, Science of Computer Programming 8 (1987) 231 -- 274.

[gupta96] V. Gupta, R. Jagadeesan, V. Saraswat, Models of concurrent constraint programming, in: Proceedings of the International Conference on Concurrency Theory, Lecture Notes in Computer Science, Vol. 1119, October 1996.

[bernard99] D. Bernard, et al., Design of the remote agent experiment for spacecraft autonomy, in: Proceedings of the IEEE Aerospace Conference, 1999.

[ingham01] M. Ingham, et al., Autonomous sequencing and model-based fault protection for space interferometry, in: Proceedings of ISAIRAS-01, 2001.

[kurien98] J. Kurien, P. Nayak, B. C. Williams, Model-based autonomy for robust mars operations, in: Proceedings of the First International Conference of the Mars Society, 1998.

[williams01] B. C. Williams, S. Chung, V. Gupta, Mode estimation of model-based programs: Monitoring systems with complex behavior, in: Proceedings of IJCAI-01, 2001.

[muscettola98] N. Muscettola, P. Nayak, B. Pell, B. C. Williams, The New Millennium Remote Agent: To boldly go where no AI system has gone before, Artificial Intelligence 103~(1-2) (1998) 5--48.

[gat96] E. Gat, ESL: a language for supporting robust plan execution in embedded autonomous agents, in: AAAI Fall Symposium on Plan Execution, 1996.

[williams97] B. C. Williams, P. Nayak, A reactive planner for a model-based executive, in: Proceedings of IJCAI-97, 1997.

[ingham01b] M. Ingham, R. Ragno, B. C. Williams, A reactive model-based programming language for robotic space explorers, in: Proceedings of ISAIRAS-01, 2001.

[williams02] B.C. Williams, M. Ingham, S. Chung, P. Elliott, Model-based Programming of Intelligent Embedded Systems and Robotic Space Explorers, submitted to: IEEE Proceedings Special Issue on Embedded Software, May 2002.