SRI INTERNATIONAL

CSL Technical Report • 03 2003

Maude Specification of the MDS Architecture and Examples

Grit Denker, Carolyn Talcott

Computer Science Laboratory

SRI-CSL-03-03, October 1, 2003

Supported by National Science Foundation under Contract CCR-0234462.

333 Ravenswood Avenue • Menlo Park, CA 94025-3493 • (650) 859-2000

Abstract

Deep space missions are very expensive and involve complex and highly customized remote agent systems such as rovers or solar airplanes. Errors that surface during a mission usually imply major financial losses and in the worst case can result in a mission's failure. Methods that provide higher assurance of correctness, predictable behavior, and greater dependability of mission software are needed to reduce the risk of costly failure. The Mission Data System (MDS) has identified two key ideas for developing a remote agent system that will simplify and reduce the cost of design, test, and operation: (1) a state-based approach to system design; and (2) a goal-oriented approach to operation. The system state is the basis on which decisions about mission operations are based. Mission goals describe the desired outcome of a mission operation so that the overall mission will be successfully completed. Goal-oriented operation allows tasks to be described in terms of "what" rather than "how." High-level goals are elaborated into flexible goal nets that combine lower-level goals.

The *Formal Checklists for Remote Agents Dependability* project will develop a formal basis, tools, and methods for enhancing dependability of goal-based operation of space missions. This includes a formal executable semantics for goal nets that is parametric in the domain model and can be instantiated to a wide range of domains; checklists of analysis tasks that assist in measuring the degree of dependability of a goal net; tools to mechanize and support the analysis tasks; and case studies for selected domain models to validate the formal framework and serve as a guide to its use. The resulting formal framework will allow certification to be added to reusability. Certified packages of goals, goal nets and corresponding software modules developed for one mission can be reused in future missions, supporting one of the objectives of remote agent architectures such as MDS that deep space missions are no longer one-of-a-kind, expensive engineering projects. This way mission systems not only become cheaper but also more reliable and the overall level of assurance of a mission's success is increased.

Contents

1 System Dependability **1**

1.1 The NSF HDCCSR program 1

1.2 The Mission Data System 1

1.3 SRI project: "Formal Checklists for Remote Agent Dependability" 2

1.4 Organization of the Report 3

2 Specification of MDS Architecture **4**

2.1 MDS components 4

2.1.1 Rewriting Logic and Maude Basics 6

2.2 Maude Modules for Architecture Components 10

2.2.1 Attributes 11

2.2.2 Interfaces 13

2.3 Maude Specification of State Variable 17

2.4 Maude Specification of Controller 23

2.5 Maude Specification of Actuator 28

2.6 Maude Specification of Device 29

2.7 Maude Specification of Sensor 30

2.8 Maude Specification of Estimator 31

3 Maude Specificaton of a Simple Rover 33

3.1 The Grid 33
3.2 The Position and Heading State Variable 35
3.3 Refined Interfaces in the Rover System 36
3.4 The Position and Heading Controller 38
3.5 The Position and Heading Actuator 42
3.6 The Rover 43
3.7 The Position and Heading Sensor 47
3.8 The Position and Heading Estimator 47

4 Analysis of the Rover System 49

4.1 An Example Scenario 49
4.2 Some Example Runs 53
4.2.1 Using Maude to run the Rover Example 53
4.2.2 Default Execution of the Rover Scenario 56
4.2.3 Search, Model-checking and Other Options 59
4.3 Lessons Learned from First Analyses 60

A Maude Specification 66

A.1 MDS Architecture 66
A.1.1 Interfaces 66
A.1.2 State Variable 71
A.1.3 Actuator 74
A.1.4 Sensor 75
A.1.5 Device 77
A.1.6 Controller 78

A.1.7 Estimator 82
A.1.8 The Grid 84
A.2 Rover System 85
A.2.1 Position and Heading Interfaces 85
A.2.2 Position and Heading State Variable 88
A.2.3 Position and Heading Controller 89
A.2.4 Position and Heading Actuator 93
A.2.5 Position and Heading Sensor 93
A.2.6 Position and Heading Device / Rover 94
A.2.7 Rover System 97

Chapter 1

System Dependability

1.1 The NSF HDCCSR program

In the context of the NSF Highly Dependable Computing and Communication Systems Research (HDCCSR) program research is funded that "promotes the ability to design, test, implement, evolve, and certify highly dependable software-based systems" (see `http://www.nsf.gov/pubs/2002/nsf02114.htm`). The program goal is "to develop a scientific basis for measurable and predictable dependability in software-based computing and communication systems, and a scientific basis—comparable to those in physics-based engineering disciplines—for technologies or methodologies to improve dependability in these systems."

NASA is providing test-bed facilities for this program. The Mission Data System (MDS) is one of these test-beds.

1.2 The Mission Data System

For several years now, NASA has been flying robotic deep space missions that rely on software to perform mission functions. Deep space missions involve a tight integration of physical and software systems. These systems are complex and expensive to design, build, and deploy. The Mission Data System

(MDS) framework has been developed to address this problem. MDS provides an architecture, tools, and libraries of reusable components to be used in the design and implementation of space mission systems, principally for robotic deep space missions that require autonomous distributed monitoring and control of physical systems.

MDS is built on two key ideas: a state-based approach to system design; and a goal-oriented approach to operation. The state-based approach makes domain knowledge explicit in the form of globally shared state variables and domain models specifying constraints on and operations for controlling the value of state variables. Goal-oriented operation allows tasks to be described in terms of *what* rather than *how.* A goal is a constraint on some state variable that is to hold over some time interval. Estimators use reports from sensor, model, and command histories to estimate the value of state variables. It is important to explicitly represent uncertainty in estimates. Controllers are responsible for satisfying primitive constraints, and must report success or a reason for failure. Goal-achiever software has the responsibility of elaborating goals into goal nets that describe actions to be carried out along with constraints on their time and order of execution. Dependability and correctness of goal achievers, and their predictable behavior in composition with concurrent activities is crucial to mission success.

1.3 SRI project: "Formal Checklists for Remote Agent Dependability"

The proposed research will develop a formal framework with methods and supporting tools for increasing the dependability of goal based operation of space systems. In particular, a formal approach to the analysis of goal net specifications is proposed that enables assertions to be made about their dependability level. A set of formal checklists (formal analysis suites) will be produced along with supporting tools that can be used to achieve more predictable dependability of goal nets and goal-based operation. The checklists will provide a qualitative means of measuring dependability of goal achievers.

The framework will be based on a formal executable semantics for goal nets that includes a formal semantics of the underlying state and domain model.

A wide spectrum of analysis techniques of different strengths will be developed to allow for achieving different levels of dependability. To experimentally validate the ideas, the framework will be applied to goal achievers for a representative set of domains in the context of the MDS testbed. The experimental work will guide the development of formal checklists. The resulting case studies will also serve as templates for further application of the formal framework.

1.4 Organization of the Report

The first step in achieving our goals is the development of a formal model of the essential features of the MDS architecture such as state knowledge, state control, state determination, behavior models, and their interactions. We use formal executable models of concurrent systems for this. In particular, we are employing rewriting logic to model MDS. The resulting system components are building blocks that are composed and further refined while specifying mission-specific applications. The MDS architecture components and their interaction patterns are defined in Chapter 2.

In Chapter 3 we illustrate how these components are further refined by filling in details for a simplified version of the SCRover.

Methods to analyze emergent properties and to check for possibilities of interference between control systems such as deadlocks and races are introduced in Section 4.

Chapter 2

Specification of MDS Architecture

2.1 MDS components

The Mission Data System (MDS) [DRRS00] and its precursor remote agent architectures [MPPW98, PBC+98] have identified two key ideas for developing a remote agent system that will simplify and reduce the cost of design, test, and operation:

1. A *state-based* approach to system design
2. A *goal-oriented* approach to operation

The system state is the basis on which decisions about mission operations are based. System state includes device operating and failure modes, device health, resource levels, and information about dynamics such as vehicle position and attitude, angles, and wheel rotation. It also includes information about the external environment such as light, terrain, temperature, and pressure. The system state is accessed through centrally managed state variables. Each state variable has associated a state time line that includes its history, current and past estimates, and future predictions and plans. In addition, there are models of state behavior that describe how a system's state evolves. Given a model and evidence data, possibly collected through measurements,

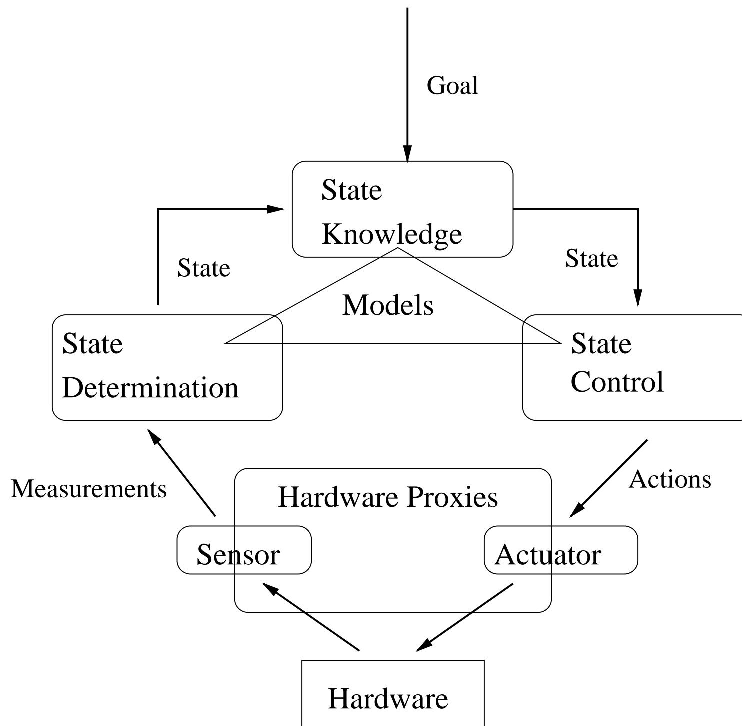

Figure 2.1: The MDS state-based architecture

the current state of the system can be explained. In conjunction with a set of proposed timed actions and the state time line, state knowledge can be propagated into the future. The combination of state and models are the basis for system operation: predicting future states, controlling toward desired states, and assessing performance.

Mission goals describe the desired outcome of a mission operation so that the overall mission will be successfully completed. Goal-oriented operation allows tasks to be described in terms of "what" rather than "how." From a mission operator's viewpoint it is a much more intuitive way to interact with the system. Mathematically, a goal is a prioritized constraint on the value of a state variable during a time interval.

The main ingredients of the MDS system are depicted in Figure 2.1 (taken from [DRRS00], with slight modifications).

In MDS all state information is held a set of so-called *state variables.* All aspects of system state that are used to contol the system must be made explict as values of state variables. State variables are the only way to access a system state. A key part of system design is the choice of state variables. State variables hold state values, degree of uncertainty and other information. *State timelines* are a complete record of a system's history. State timelines capture both knowledge (i.e., estimated states from the past) and intents (i.e., planned state of the future). Domain knowledge is expressed separately in *models.* These models express constraints on values of state variables and predict how they will change under given actions (phyics, device specs). State variables receive primitive goals from a scheduler. A goal is a constraint on the value of a state variable over a time interval. The goals are forwarded to the *controller* of the state variable. In order to satisfy the goal, controllers issue commands on the basis of the current value of the state variable. The commands are transmitted via *actuators* in the hardware proxies to the hardware. *Sensors* convey measurements from the hardware via the proxy to *estimators* (also called state determination in Figure 2.1). Estimators interpret measurements and update values of state variables. This closes the cycle. The updated value of the state variable can be compared against the goal and further steps can be taken to either achieve a goal that failed or to tackle the next goal.

2.1.1 Rewriting Logic and Maude Basics

Rewriting logic [Mes92, Mes00] is a very simple logic in which a distributed system can be formally specified by giving an algebraic data type axiomatizing a system's distributed state space, and a collection of rewrite rules specifying the *dynamics* of the system in terms of local transitions. The algebraic data type is given means of an equational specification consisting of a signature of types and operations Σ and a collection of conditional equations E. A rewrite rule has the form

$$t \to t'$$

where t, t' are Σ-terms, that describe the *local, concurrent transitions* possible in the system, namely, when a part of the distributed state fits the pattern t, then it can change to a new local state fitting the pattern t'. Therefore,

a *rewrite theory* is a triple (Σ, E, R), with (Σ, E) an equational specification axiomatizing a system's distributed state space, and R a collection of rewrite rules axiomatizing the system's local transitions.

Maude [CELM96, CDE+99] is a multi-paradigm executable specification language based on rewriting logic. The Maude interpreter is very efficient, allowing prototyping of quite complex test cases. The reflective capabilities of rewriting logic and Maude[CM96] allow user-defined execution strategies to be formally specified by rewrite rules at the meta level. This allows easy exploration of state spaces of interest, including strategies such as breadth-first search that can exhaustively explore all executions, up to some depth, of a system from a given initial state. Maude provides efficient built-in search and model checking capabilities. The search facility can be used to find states matching some pattern reachable from a given initial state, and to produce a transition graph of the state space explored. The model checker can be used to test an initial state for satisfaction of formulae in linear temporal logic. A counterexample is returned if the checker finds that the formula is not satisfied. Maude sources, executables for several platforms, the manual, a primer, cases studies and papers are available from the Maude website `http://maude.cs.uiuc.edu` or its mirror `http://maude.csl.sri.com`.

We briefly summarize the syntax of Maude. We use two types of modules:

- *functional* modules, that are equational theories used to specify algebraic data types; they are declared with the syntax `fmod ... endfm` and
- *system* modules, that are rewrite theories specifying concurrent systems; they are declared with the syntax `mod ... endm`

The above modules can be *parameterized*, and have an *initial model semantics*. That is, they specify the initial models of their corresponding theories, or, in the case of parameterized modules, the free extension of a model of the parameter theory to a model of the body.

Immediately after the module's keyword, the *name* of the module is given. After this, a list of imported submodules can be added. One can also declare *sorts* and *subsorts* and *operators*. Operators are introduced with the `op` keyword. They can have user-definable syntax, with underbars '_' marking

the argument positions, and are declared with the sorts of their arguments and the sort of their result. Some operators can have equational *attributes*, such as `assoc` and `comm`, stating, for example, that the operator is associative and commutative. Such attributes are then used by the Maude engine to match terms *modulo* the declared axioms.

There are three kinds of logical axioms, namely, *equations*—introduced with the keywords `eq`, or, for conditional equations, `ceq`—*memberships*—declaring that a term has a certain sort and introduced with the keywords `mb` and `cmb`—and *rewrite rules*—introduced with the keywords `rl` and `crl`. Functional modules can only have equations and memberships. System modules can have any of the three kinds of axioms. The mathematical variables in such axioms are declared with the keywords `var` and `vars`.

In this report, given that a concurrent object-oriented style fits remote agent systems very well, we will focus on the specification of concurrent object-oriented systems. There are various language features that support object-oriented specification in Maude. One of the pre-defined modules of Maude acilitates object-oriented specification, it is the system module `CONFIGURATION`.

```
mod CONFIGURATION is
  sorts Attribute AttributeSet .
  subsort Attribute < AttributeSet .
  op none : -> AttributeSet .
  op _,_ : AttributeSet AttributeSet -> AttributeSet
           [ctor assoc comm id: none] .

  sorts Oid Cid Object Msg Configuration .
  subsort Object Msg < Configuration .
  op <_:_|_> : Oid Cid AttributeSet -> Object [ctor object] .
  op none : -> Configuration .
  op __ : Configuration Configuration -> Configuration
          [ctor config assoc comm id: none] .
endm
```

Objects typically have several attributes. For this reason the sorts `Attribute` and `AttributeSet` are defined. A constructor `none` is provided for empty attribute sets and another constructor for composing attribute sets is defined as well.

Objects are identified by so-called *object identifiers* for which the sort `Oid` is defined. Objects are instances of classes, identified by *class identifiers* of sort `Cid`. The object-oriented approch of Maude understands a (snapshot of a) system state (`sort Configuration`) as a multiset of objects (`sort Object`) and messages (`sort Msg`). Typically, a configuration is a multiset of objects and messages. The multiset union operator for configurations is associative and commutative so that order and parentheses do not matter, and so that rewriting is multiset rewriting in Maude. It is denoted with empty syntax (juxtaposition). Declaring the sorts `Object` and `Msg` as subsorts of `Configuration` allows to build system configurations starting from single objects or messages.

A notation for object syntax is also defined. An object is defined by its identifier, its class and a set of attributes. Thus, objects of a class are then record-like structures of the form

$$< O; :C \mid a_1 , \ldots , a_n >$$

with O an object identifier, C an class identifier, and $a_1, \ldots, a_n$ the names of the object's attributes. It is up to the designer of a system to define appropriate attributes with corresponding domain and range sorts. Examples will be given in the following section.

Objects can interact with each other in a variety of ways, including the sending of messages. Maude does not predefine message formats. For the purpose of the MDS application, we defined the following module defining for object identity and message constructors.

```
mod MYCONF is
  inc CONFIGURATION .
  inc STRING .

  op o : String  -> Oid .

  sort MsgBody .
  op msg : Oid Oid MsgBody -> Msg .

  op noMsg : -> Msg .
     *** needed for intial configurations
endm
```

As for object identifiers, we declared a constructor `o` that turns a string into an object identifier. A message is constructed with three parameters, the first two ones of type `Oid` referring to the receiver and the sender, respectively, of the message, and the last parameter of sort `MsgBody`, holding the content of the message.

A typical system configuration will consist of several objects and messages. The dynamic behavior of a concurrent object system is then axiomatized by specifying each of its basic concurrent transition patterns by a corresponding labeled rewrite rule.

2.2 Maude Modules for Architecture Components

Our formal model is based on publications describing the MDS architecture [DRRS00], documents available from the SCRover testbed artifact website [BGH+03c, BGH+03b, BGH+03a] (including architectural specifications, UML diagrams of the interactions between the various components in attempting to satisfy the goal, and C++ header files) and several conversations with members of the MDS team at JPL and the SCRover team at USC. In particular a one day meeting at USC with the group developing the SCRover Testbed clarified many details regarding both the SCRover (hardware characteristics and capabilities) and the MDS framework (especially issues regarding timing and the interface to the scheduler).

We model the MDS architecture with the following six main modules, each defining a class: `STATE-VARIABLE, CTRL, ESTIMATOR, ACTUATOR, SENSOR,` and `DEVICE`. These classes correspond to the MDS components *State Knowledge* (or state variable), *State Control* (or controller), *State Determination* (or estimator), *Actuator*, *Sensor*, and *Hardware* (or device). In which ways these components are connected and which component is connected to which other component via communication is depicted in Figure 2.2.

The state variable knows about its controller and estimator, and vice a versa. A state variable has exactly one estimator and one controller. An estimator may compute estimates for more than one state variable. Similarly, a controller may control more than one state variable.

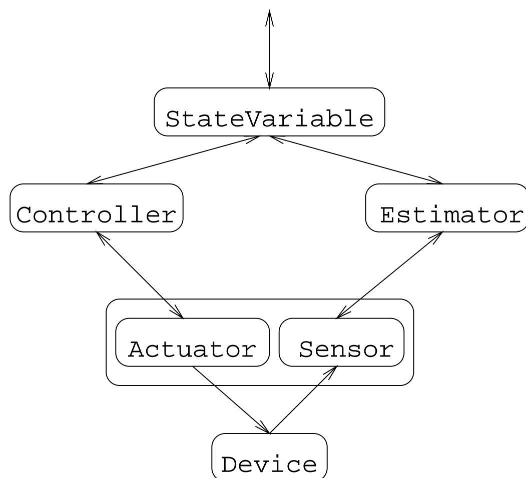

Figure 2.2: MDS Component Communication

The information and control flow is roughly as follows: A goal request is made to the state variable, which forwards it to the controller. The controller decides which actions to take, determines the course of action and issues commands for the device via the actuator. The device sends the latest sensor values to the estimator. The estimator creates measurements out of those values and sends update requests to the state variable. When the state variable receives and update, it acknowledges this with a message to the estimator and informs its controller about the new value. The controller can check the new state against the goal and decide further actions. It either continues issuing commands to the actuator in order to achieve the goal, or it reports back to the state variable whether the goal was achieved successfully or failed. The state variable forwards the information to the environment that started the constraint (goal) in the first place.

2.2.1 Attributes

As the above description lays out, the components have references to some of the other components. For example, the state variable has a reference to its estimator and its controller, and the controller has a refernce to its state variable and the actuator. These references are specified as attributes

of objects in the correponding classes. For simplicity, we defined one module `ATTRIBUTES` that defines all possible references to other objects. This module is to be imported in all other module. Nevertheless, the intended use is that the state variable only has an attribute that refers to its controller and another one that refer to its estimator, but it does not have references to other components. Similarly, the controller knows its actuator and state variable (and vice a versa), the estimator knows its state variable and its sensor (and vice a versa), the device knows both its actuator and its sensor (and vice a versa). This intended use could have been reflected in a more complicated module structure, but we did not thought of that essential and chose the simpler module structure.

```
fmod ATTRIBUTES is
  inc MYCONF .

  op myest  : Oid -> Attribute .
  op mysv  : Oid -> Attribute .
  op myctrl  : Oid -> Attribute .
  op mydevice  : Oid -> Attribute .
  op myactuator : Oid -> Attribute .
  op mysensor  : Oid -> Attribute .

  op waitAfter : Msg -> Attribute .
     *** attribute for control-flow within an object
     *** contains the last message received
     *** if there was no last message received,
     *** then it contains the current message sent
endfm
```

A special attribute called `waitAfter` that takes a message as parameter has been specified. This attribute is used in some of the components to restrict the object's behavior in a sequential way. If used, this attribute contains the last message that has been sent or received. In a rule one can require that the attribute contains a certain message, assuring that the object can only proceed *after* it has sent or received a certain message. Generally, an object may *fire* (apply) any rule, as long as the current system configuration matches the lefthand side of the rule. This way, Maude is inherently concurrent, allowing several objects to apply several rules at the same time, as

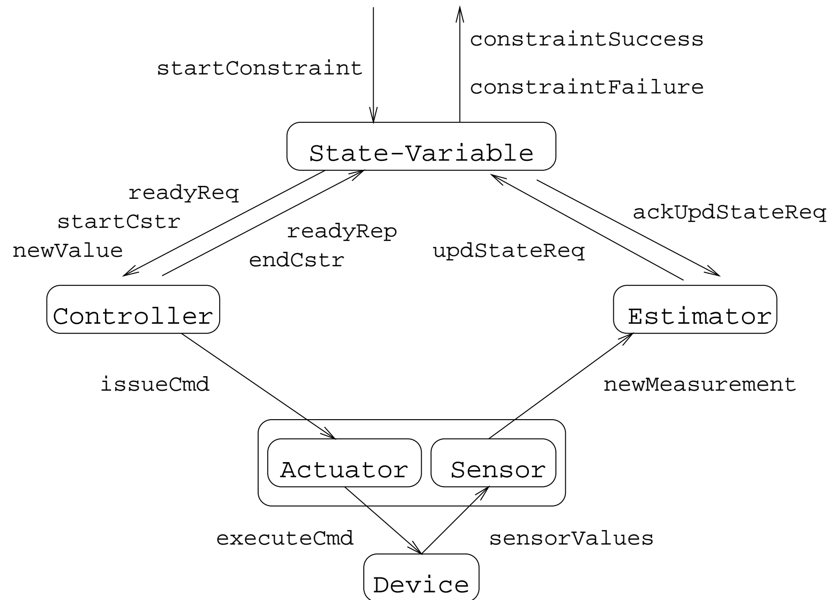

Figure 2.3: MDS Component Message Exchange

long as the system configuration matches the combination of all the lefthand sides of the rules in question. Moreover, Maude specifications is inherently non-deterministic, as an object may non-deterministically choose which rule to execute, in case more than one rule is applicable. Using the waitAfter attribute can force objects to behave sequentially.

2.2.2 Interfaces

All message exchanges between components in the MDS architecture are asynchronous. Some of the message exchanges are meant to model function calls as known from conventional programming. To simulate that the calling object is blocked until the call returns, for which we used acknowledgement messages in Maude. Figure 2.3 gives an overview of the kinds messages that are sent between components and about the direction of message flow.

The messages at the interface between two components need to be available and specified for both components. To avoid redundant declaration, we de-

fine interface modules for all seven interfaces, between (a) environment and the state variable, (b) state variable and controller, (c) state variable and estimator, (d) controller and actuator, (e) actuator and device, (f) device and sensor, and (g) sensor and estimator.

State Variable/Evironment Interface. The environment starts a constraint with the state variable. The state variable reports back to the environment whether the constraint could be achieved or not. In case of failure, it will deliver a reason.

We start with defining sorts for constraints and reasons together with constructor operations. The constant `noCstr` is used to indicate that there is no constraint, that is its the constraint that is always satisfied, analogous to the boolean value `true`.

```
fmod CONSTRAINT is
  sort Constraint .
  op noCstr : -> Constraint .
endfm

fmod REASON is
  sort Reason .
  op noReason : -> Reason .
endfm
```

Now we can define the messages for the interface between state variables and the environment, importing all other necessary modules.

```
fmod STATE-VARIABLE-ENVIRONMENT-INTERFACE is
  inc MYCONF .
  inc CONSTRAINT .
  inc REASON .

  op startConstraint : Constraint -> MsgBody .
  op constraintSuccess : Constraint -> MsgBody .
  op constraintFailure : Constraint Reason -> MsgBody .
```

```
                              *** Reason - for failure
endfm
```

State Variable/Controller Interface. The state variable checks whether the controller is ready for a new constraint. If it receives a positive reply, it will forward the constraint, including the requestor identity (from the environment) and its current state value. Thus, we need to define a sort for state values. The value of a state variable maybe be uncertain and in particular it may simply be unknown.

```
fmod STATE-VALUE is
  sort StateValue .

  sort UnknownStateValue .
  subsort UnknownStateValue < StateValue .

  op uk : -> UnknownStateValue .
endfm
```

For the time being we restrict ourselves to a simple state value sort and a subsort denoting undefined state values for which a constructor is given. In the future we will refine this specification to include the state values used in the MDS system (such as RTEpoch etc.).

The interface module for controllers and state variables define formats for potential message contents.

```
fmod STATE-VARIABLE-CONTROLLER-INTERFACE is
  inc MYCONF .
  inc CONSTRAINT .
  inc REASON .
  inc STATE-VALUE .

  op readyReq : -> MsgBody .
  op readyRep : Bool -> MsgBody .

  op startCstr : Constraint Oid StateValue -> MsgBody .
```

```
  op endCstr : Constraint Oid Bool Reason -> MsgBody .

  op newVal : StateValue -> MsgBody .
endfm
```

Controller/Actuator Interface. The controller issues commands to the actuator. For this purpose, we define a sort `command`.

```
fmod COMMAND is
  sort Command .
  op noCmd : -> Command .
endfm

%\todo{CLT: is noCmd used?}

fmod CONTROLLER-ACTUATOR-INTERFACE is
  inc MYCONF .
  inc COMMAND .
  op issueCmd : Command -> MsgBody .
endfm
```

Device/Actuator Interface. Devices execute the commands sent to them by the actuator.

```
fmod DEVICE-ACTUATOR-INTERFACE is
  inc MYCONF .
  inc COMMAND .

  op executeCmd : Command -> MsgBody .
endfm
```

Device/Sensor Interface. After executing the command, new values are sent to the sensor.

```
fmod SENSOR-VALUE is
```

```
  sort SensorValue .
endfm

fmod DEVICE-SENSOR-INTERFACE is
  inc MYCONF .
  inc SENSOR-VALUE .

  op sensorValues : SensorValue -> MsgBody .
endfm
```

Sensor/Estimator Interface. The Sensor turns the new values into a measurement that is forwarded to the estimator.

```
fmod MEASUREMENT is
  sort Measurement .
  op  noMeas : -> Measurement .
endfm

fmod ESTIMATOR-SENSOR-INTERFACE is
  inc MYCONF .
  inc MEASUREMENT .

  op newMeasurement : Measurement -> MsgBody .
endfm
```

2.3 Maude Specification of State Variable

The `STATE-VARIABLE` module includes the necessary modules for attributes and interfaces with the controller and estimator. It defines the state variable class identifier `SVCid` and makes it a subsort of the predefined `Cid` defined in `CONFIGURATION`. This way the state variable class is a subclass of the generic class. Subclassing is an essential feature of object-oriented system design since it allow reuse of classes and behavior. Later we will define a special state variable such as a position state variable, as a subclass of `SVCid`. This way, all rules defined for state variables will apply to all specialized state variables.

We define additional attributes for state variables. State variables have a current value, the constraint that is being processed, if any, and an object identifier to remember the requester of that constraint.

In case the controller is busy processing an older constraint (or goal), the state variable can, after communicating with the controller, report to the environment that it is not ready for a new constraint. A `notReady` constructor for the sort `Reason` has been defined for those situations.

```
mod STATE-VARIABLE is
  inc ATTRIBUTES .
  inc STATE-VARIABLE-ENVIRONMENT-INTERFACE .
  inc STATE-VARIABLE-CONTROLLER-INTERFACE .
  inc STATE-VARIABLE-ESTIMATOR-INTERFACE .

  sort SVCid .
  subsort SVCid < Cid .

  *** Attributes
  op  val : StateValue -> Attribute .
  op  req : Oid -> Attribute .
  op  cstr :  Constraint -> Attribute .

  *** Reasons
  op notReady : -> Reason .
```

The set of rules defined within the state variable module defines the behavior of state variables. The rules are defined using variables of appropriate types. This way the rule template can be matched by specific state variable instances.

```
  vars sv o o' ctrl : Oid .
  var svcid : SVCid .
  var cstr cstr' : Constraint .

  rl[startConstraintReadyReq]:
  < sv : svcid | myctrl(ctrl),
                 req(o'),
```

```
                cstr(cstr'),
                svatts:AttributeSet >
msg(sv,o,startConstraint(cstr))
=>
< sv : svcid | myctrl(ctrl),
                req(o),
                cstr(cstr),
                svatts:AttributeSet >
msg(ctrl,sv,readyReq)  .
```

This rule expresses the result of asynchronous receipt of a message

```
msg(sv,o,startConstraint(cstr))
```

by an object of class `SVCid`. The message from an object *o* in the environment to the state variable *sv* requests the start of a new constraint *cstr*. The message is consumed, the object changes its value of the latest requestor to *o* and also stores the latest constraint to be achieved. Moreover, it sends a message to its controller, asking whether the controller is ready to process a new message.

In general, a rewrite rule may have several objects in its lefthand side, describing a *synchronization* or *multi-party interaction* between several objects. However, we decided to model the MDS communication as asynchronous and the lefthand sides will consist of either an object and a message, or just an object. Attributes whose value do not change or do not affect the next state of other attributes are represented by the variable `svatts` of sort `AttributeSet`.

Variables can be declared separately in the variables section of a Maude module (as done for o, sv, etc.) and reused in several rules, or they can be specified with their domain every time when they are used in a rule.

The second rule for state variables determines the behavior when it receives the reply from the controller. If the controller is busy working on a previous constraint, the state variable will sent a message to the requester saying it is not ready and that the constraint failed. If the controller is ready for a new constraint, the state variable will forward the constraint along with the requester and its current state value. [1]

[1]Remark: This is a slight optimization over the UML diagram in which the controller

```
rl[forwardConstraint]:
  < sv : svcid | myctrl(ctrl),
                 req(o),
                 cstr(cstr),
                 val(v),
                 svatts:AttributeSet >
  msg(sv,ctrl,readyRep(b))
  =>
  if b
  then < sv : svcid | myctrl(ctrl),
                      req(o),
                      cstr(cstr),
                      val(v),
                      svatts:AttributeSet >
       msg(ctrl,sv,startCstr(cstr,o,v))
  else < sv : svcid | myctrl(ctrl),
                      req(o),
                      cstr(cstr),
                      val(v),
                      svatts:AttributeSet >
       msg(o,sv,constraintFailure(cstr,notReady))
  fi .
```

The state variable can receive a state value update request from its estimator. If this happens, it stores the latest state value, acknowledges the receipt of the update request to the estimator and informs the controller about the newest value.

```
  rl[stateValUpdNotifyCtrlAckEst] :
  < sv : svcid | myctrl(ctrl),
                 myest(est),
                 val(v),
                 svatts:AttributeSet >
  msg(sv,est,updStateReq(v2))
  =>
  < sv : svcid | myctrl(ctrl),
```

requests the latest state value. The value v is assumed to be the current best estimate.

```
                 myest(est),
                 val(v2),
                 svatts:AttributeSet >
  msg(ctrl,sv,newVal(v2))
  msg(est,sv,ackUpdStateReq(v2)) .
```

The controller must eventually finish processing a constraint either by determining that it has succeeded or by declaring failure. It then sends an `endConstraint` message to its the state variable. In case of success, the message contains the constraint that has been achieved along with the requestor who originated that request. The parameter for the reason is set to `noReason`, indicating that the constraint was successfully achieved. In case of failure, the message also contains a reason for failure.

```
  rl[endConstraint] :
  < sv : svcid | myctrl(ctrl),
                 svatts:AttributeSet >
  msg(sv,ctrl,endCstr(cstr,o,r))
  =>
  if (r=noReason)
  then < sv : svcid | myctrl(ctrl),
                      svatts:AttributeSet >
       msg(o,sv,constraintSuccess(cstr))
  else < sv : svcid | myctrl(ctrl),
                      svatts:AttributeSet >
       msg(o,sv,constraintFailure(cstr,r))
  fi .
endm
```

To illustrate how a particular instance of a state variable can be defined, we include a module `STATE-VARIABLE-TEST` that defines `mystatevar`, an object of class `StateVar`.

```
mod STATE-VARIABLE-TEST is
  inc STATE-VARIABLE .

  op StateVar : -> SVCid .
```

```
  op mystatevar : -> Object .
  eq mystatevar = < o("MyStateVar") : StateVar |
                       myctrl(o("MyCtrl")),
                       myctrl(o("MyEstimator")),
                       req(o("noOid")),
                       cstr(noCstr),
                       val(uk) > .
endm
```

A constant of type `SVCid` has been defined. One could question why we did not define this constant in the `STATE-VARIABLE` module and used `StateVar` in all the rules, e.g.,

```
rl[startConstraintReadyReq]:
  < sv : StateVar | myctrl(ctrl),
                    req(o'),
                    cstr(cstr'),
                    svatts:AttributeSet >
  msg(sv,o,startConstraint(cstr))
  =>
  < sv : StateVar | myctrl(ctrl),
                    req(o),
                    cstr(cstr),
                    svatts:AttributeSet >
  msg(ctrl,sv,readyReq) .
```

The reason for not using the constant is, that the rules would only be applicable to those objects that match exactly this constant. Therfore, when we later define a specialized class of state variables, such as a position state variable class, the rules would not apply to instances of the specialized class. Thus, by choosing to model the rules of the state variable class using variables for the object identifier as well as the class identifier, enables inheritance. A position state variable class `posSVCid` can be defined as a subsort of `SVCid` and the rules for state variables will also be matched by position state variables.

2.4 Maude Specification of Controller

In this version of the MDS architecture specification we specify a controller with a generic control strategy. The controller determines the course of action to satisfy a goal and issues appropriate commands. The list of commands to be processed is stored in a `CommandList`. The corresponding datatype is specified below (including operations to sequentially access elements of the list).

```
fmod COMMAND-LIST is
  inc COMMAND .
  sort CommandList .
  subsort Command < CommandList .

  op nil : -> CommandList .
  op _;_ : CommandList CommandList -> CommandList
           [ctor assoc id: nil] .
  op first : CommandList -> Command .
  op rest : CommandList -> CommandList .

  var cmdl : CommandList .
  var cmd : Command .

  eq first(nil) = noCmd .
  eq first(cmd ; cmdl) = cmd .
  eq rest(nil) = noCmd .
  eq rest(cmd ; cmdl) = cmdl .
endfm
```

Similar to the state variable, the controller module imports the necessary attribute, interface and datatype modules. A controller class id sort is defined as subsort to the generic class identity sort.

A controller keeps the constraint that is to be achieved in an attribute, along with attributes for its requestor, the current state variable value, and the list of commands to be executed.

```
mod CONTROLLER is
```

```
inc ATTRIBUTES .
inc STATE-VARIABLE-CONTROLLER-INTERFACE .
inc CONTROLLER-ACTUATOR-INTERFACE .
inc COMMAND-LIST .

sort CtrlCid .
subsort CtrlCid < Cid .


*** Attributes
op  currentCstr : Constraint -> Attribute .
    *** constraint that is currently handled
op  currentCstrReq : Oid -> Attribute .
    *** requester who started current constraint
op  currentSVVal : StateValue -> Attribute .
    *** last requested state value of state variable
op cmds : CommandList -> Attribute .
```

The controller decides whether the current state value satisfies the constraint using a `satisfy` operation. This operation will be defined separately for each particular application of the MDS architecture. For example, when a rover is supposed to achieve a certain position on a map, the satisfy operation will be refined to determine whether a given position corresponds to the requested position.

On the basis of the state value and the constraint, the controller will determine a course of actions, consisting of a list of commands. In case the controller has executed all the commands and still not achieved the goal, it will report back to the state variable with the reason `COANoSuccess`, saying that the course of action determined by the controller was not successful.

```
*** Internal computations
op satisfy : StateValue Constraint -> Bool .
   *** tests whether current state satisfies constraint
op coa : StateValue Constraint -> CommandList .
   *** determines course of action,
   *** given current value and constraint
```

```
*** Reason for failure:
*** course of action has been executed, constraint still
*** not satisfied
op COANoSuccess : -> Reason .
```

When the controller receives a request from the state variable for a new goal, it will respond with a positive reply, if it is not in the process of achieving a constraint (waiting for an accepted constraint or in the cycle of issuing commands as indicated `waitAfter(noMsg)`). Otherwise it will respond negatively.

```
rl[readyForNewConstraint]:
< c : ccid | mysv(sv),
             waitAfter(msg),
             ctrlatts:AttributeSet >
msg(c,sv,readyReq)
=>
if (msg == noMsg)
then < c : ccid | mysv(sv),
                  waitAfter(msg),
                  ctrlatts:AttributeSet >
     msg(sv,c,readyRep(true))
else < c : ccid | mysv(sv),
                  waitAfter(msg),
                  ctrlatts:AttributeSet >
     msg(sv,c,readyRep(false))
fi .
```

When the controller starts a new constraint by asynchronously receiving a message from the state variable, it saves the constraint, the requestor, and the current state value. If the current state value already satisfies the constraint, it replies to the state variable with a sucessful `endCstr` message. If the current state does not fulfill the constraint, the controller determines the course of action by calling the internal `coa` function that returns a list of commands, stores the tail of the command list for future actions and sends a message to the actuator issuing the first command in the command list.

```
rl[getCstrCompareGoalIssueCommand]:
< c : ccid | mysv(sv),
             myactuator(a),
             currentCstr(cstr'),
             currentCstrReq(o'),
             currentSVVal(v'),
             cmds(cl),
             waitAfter(noMsg),
             ctrlatts:AttributeSet >
msg(c,sv,startCstr(cstr,o,v))
=>
if satisfy(v,cstr)
then < c : ccid | mysv(sv),
                  myactuator(a),
                  currentCstr(cstr),
                  currentCstrReq(o),
                  currentSVVal(v),
                  cmds(cl),
                  waitAfter(noMsg),
                  ctrlatts:AttributeSet >
     msg(sv,c,endCstr(cstr,o,noReason))
else < c : ccid | mysv(sv),
                  myactuator(a),
                  currentCstr(cstr),
                  currentCstrReq(o),
                  currentSVVal(v),
                  cmds(rest(coa(v,cstr))),
                  waitAfter(msg(c,sv,startCstr(cstr,o,v))),
                  ctrlatts:AttributeSet >
     msg(a,c,issueCmd(first(coa(v,cstr))))
fi .
```

Once the state variable reports a new value to the controller, for example because the sensor sensed new values as a result of executing the command via the actuator, and the estimator updated the state variable accordingly, then the controller will retest satisfaction of the constraint. If the constraint is satisfied, it replies to the state variable with an appropriate successful `endCstr` message. It sets its `waitAfter` attribute back to `noMsg` to allow

new constraints. If the constraint is not satisfied, but the controller has not more commands to issue, it will report to the state variable accordingly. Only if there are commands left to be issued, the controller will do so and keep `waitAfter` attribute set to the message requesting the current constraint to block further request for the time being.

```
rl[getNewValueCompareGoal]:
< c : ccid | mysv(sv),
             myactuator(a),
             currentCstr(cstr),
             currentCstrReq(o),
             currentSVVal(v),
             cmds(cl),
             waitAfter(msg(c,sv,startCstr(cstr,o,v''))),
             ctrlatts:AttributeSet >
msg(c,sv,newVal(v'))
=>
if satisfy(v',cstr)
then < c : ccid | mysv(sv),
                  myactuator(a),
                  currentCstr(cstr),
                  currentCstrReq(o),
                  currentSVVal(v'),
                  cmds(cl),
                  waitAfter(noMsg),
                  ctrlatts:AttributeSet >
     msg(sv,c,endCstr(cstr,o,noReason))
else (if cl == nil
      then < c : ccid | mysv(sv),
                        myactuator(a),
                        currentCstr(cstr),
                        currentCstrReq(o),
                        currentSVVal(v'),
                        cmds(cl),
                        waitAfter(noMsg),
                        ctrlatts:AttributeSet >
           msg(sv,c,endCstr(cstr,o,COANoSuccess))
      else < c : ccid | mysv(sv),
```

```
                              myactuator(a),
                              currentCstr(cstr),
                              currentCstrReq(o),
                              currentSVVal(v'),
                              cmds(rest(cl)),
                              waitAfter(msg(c,sv,startCstr(cstr,o,v))),
                              ctrlatts:AttributeSet >
             msg(a,c,issueCmd(first(cl)))
        fi)
  fi .
```

2.5 Maude Specification of Actuator

The current specification of the actuator is rather simplistic. The actuator merely stores the latest command it received from the controller and forwards it to the device.

```
mod ACTUATOR is
  inc ATTRIBUTES .
  inc CONTROLLER-ACTUATOR-INTERFACE .
  inc DEVICE-ACTUATOR-INTERFACE .

  sort ActuatorCid .
  subsort ActuatorCid < Cid .


  *** Attributes
  op  executedCmd : Command -> Attribute .
      *** the command that was sent by the controller for execution

  vars act d ctrl : Oid .
  var actcid  : ActuatorCid .
  vars cmd cmd' : Command .

  rl[executeCommandReq]:
  < act : actcid | mydevice(d),
```

```
                   myctrl(ctrl),
                   executedCmd(cmd'),
                   actatts:AttributeSet >
  msg(act,ctrl,issueCmd(cmd))
  =>
  < act : actcid | mydevice(d),
                   myctrl(ctrl),
                   executedCmd(cmd),
                   actatts:AttributeSet >
  msg(d,act,executeCmd(cmd)) .
endm
```

2.6 Maude Specification of Device

The device component of the MDS architecture is external to the software. There are two reasons to include devices in the formal model. One is that this makes explict the 'physics' that goals and controllers are relying on. The other is that executable models of devices can be used in simulation and analysis of goal achiever specifications.

All of the behavior of the device depends on the specific application Therefore, the generic device module on provides a sort definition for device class identifiers and details the subsort relations with the general sort for class identities.

```
mod DEVICE is
  inc ATTRIBUTES .
  inc DEVICE-ACTUATOR-INTERFACE .
  inc DEVICE-SENSOR-INTERFACE .

  sort DeviceCid .
  subsort DeviceCid < Cid .
endm
```

2.7 Maude Specification of Sensor

The sensor receives new values from the device. Out of these values, the sensor produces new measurements that are forwarded to the estimator for further processing. Generally, we cannot assume that sensor values and measurement are the same datatypes. Therefore, the sensor provides an operation to convert sensor values into measurements.

For the time being we restricted ourselves to a sensor specification in which the sensor only stores the latest measurement. More generally, the sensor has the history of all measurements and in the future we will refine this specification to allow for measurement histories. Similarly, the state variable will contain timeliness instead of only a singleton state value, to represent all past and planned state values.

```
mod SENSOR is
  inc ATTRIBUTES .
  inc ESTIMATOR-SENSOR-INTERFACE .
  inc DEVICE-SENSOR-INTERFACE .

  sort SensorCid .
  subsort SensorCid < Cid .

  *** Attributes
  op  measurement : Measurement -> Attribute .

  *** Converts values to measurements
  op createMeasurement : SensorValue -> Measurement .

  vars d est sens : Oid .
  var  senscid  : SensorCid .
  vars sval : SensorValue .
  var  m : Measurement .

 rl[processSensorValuesSendNewMeasurementToEstimator]:
  < sens : senscid | myest(est),
                     mydevice(d),
             measurement(m),
```

```
                  sensatts:AttributeSet >
  msg(sens,d,sensorValues(sval))
  =>
  < sens : senscid | myest(est),
                     mydevice(d),
           measurement(createMeasurement(sval)),
                     sensatts:AttributeSet >
  msg(est,sens,newMeasurement(createMeasurement(sval))) .
endm
```

2.8 Maude Specification of Estimator

The estimator gets a new measurement from the sensor and turns it into the appropriate representation of a new state value. When receiving a new measurement, it will send a request for udpating the state value to the state variable. Once it has received the acknowledgement for its request, it is available for receiving further measurements. Sequentializing the estimator and waiting for the acknowledgement makes sure that no state updates are lost due to messages that overtake other messages.

```
mod ESTIMATOR is
  inc ATTRIBUTES .
  inc ESTIMATOR-SENSOR-INTERFACE .
  inc STATE-VARIABLE-ESTIMATOR-INTERFACE .

  sort EstimatorCid .
  subsort EstimatorCid < Cid .

  *** Attributes
  op state : StateValue -> Attribute .

  *** converts a measurement into a state value
  op verifyState : Measurement  -> StateValue .

  vars sens est sv : Oid .
  var estcid  : EstimatorCid .
```

```
  var m : Measurement .
  var v : StateValue .

  rl[receiveMeasurementUpdStateReq]:
  < est : estcid | mysensor(sens),
                   mysv(sv),
                   waitAfter(noMsg),
                   state(v),
                   estatts:AttributeSet >
  msg(est, sens, newMeasurement(m))
  =>
  < est : estcid | mysensor(sens),
                   mysv(sv),
                   waitAfter(msg(est,sens,newMeasurement(m))),
                   state(verifyState(m)),
                   estatts:AttributeSet >
  msg(sv,est,updStateReq(verifyState(m))) .


  rl[receiveAckUpdateStateVar]:
  < est : estcid | mysv(sv),
                   waitAfter(msg(est,sens,newMeasurement(m))),
                   estatts:AttributeSet >
  msg(est,sv,ackUpdStateReq(v))
  =>
  < est : estcid | mysv(sv),
                   waitAfter(noMsg),
                   estatts:AttributeSet > .

endm
```

This concludes the specification of all generic MDS components in Maude. In the next chapter we illustrate how we can use these modules in the designing a small remote control agent system, namely a rover navigating over a grid.

Chapter 3

Maude Specificaton of a Simple Rover

Our example of a small remote agent system is a rover that is moving on a grid along the x- and y-axes. The rover is able to turn clockwise in 45 degree units. The goal of the rover is to reach a certain position on the grid, specified via (x, y) coordinates. Some location on the grid are blocked so that the rover cannot maneuver using those positions.

3.1 The Grid

We start with specifying the grid. A location is a pair of natural numbers indicating (x, y) coordinates. Locations can be collected in a `LocationSet`.

A member function determines whether a specific location is part of a location set. This function will be called when the rover attempts to move one step, to see if the new location is a member of the blocked set. If the new location is blocked, the attempt fails and the position of the rover is not changed. If the new location is not blocked, then the rover can move onto the new location. The rover will always move in the direction in which it is oriented, which is one of North, West, South, or East. Thus, the next location of a rover is determined by its current location and its direction.

A converter function allows to associate directions to certain numbers. That

is the direction 0 (degrees) is defined to be north, the direction 90 is defined to be east, and so forth.

```
fmod GRID is
  pr NAT .

  sorts Loc LocSet .
  subsort Loc < LocSet .

  op mt : -> LocSet .
  op __ : LocSet LocSet -> LocSet [ assoc comm id: mt ] .

  op `(_`,_`) : Nat Nat -> Loc .

  op member : Loc LocSet -> Bool .
  eq member(l:Loc, l:Loc ls:LocSet) = true .
  eq member(l:Loc, ls:LocSet) = false [owise] .

  sort Dir .
  ops N W E S : -> Dir .

  op newLoc : Nat Nat Loc Dir LocSet -> Loc .
  op next : Nat Nat Loc Dir -> Loc .

  vars ht wd x y : Nat .

  eq newLoc(ht, wd, (x,y) , d:Dir, blocked:LocSet)
      = ( if member(next(ht,wd,(x,y),d:Dir),blocked:LocSet)
          then (x,y)
          else next(ht,wd,(x,y),d:Dir)
          fi )  .

  eq next(ht,wd, (x,y), N) =
       if (s y < ht) then (x, s y) else (x,y) fi .

  eq next(ht,wd, (x,y), S) =
       if (0 < y) then (x, sd(y,1)) else (x,y) fi .
```

```
  eq next(ht,wd, (x,y), E) =
       if (s x < wd) then (s x, y) else (x,y) fi .

  eq next(ht,wd, (x,y), W) =
       if (0 < x) then (sd(x,1), y) else (x,y) fi .

  op dir : Nat -> Dir .
  eq dir(0) = N .
  eq dir(90) = E .
  eq dir(180) = S .
  eq dir(270) = W .
endfm
```

3.2 The Position and Heading State Variable

As discussed above, an important part of designing and MDS system is identifying the state variables, how they are to be measured and controlled. Goals of the rover system include moving to a certain location on the grid and facing in a given direction. Thus, we define a class for position and heading state variables in the module `POSANDHEAD-STATE-VARIABLE`. To assure that all rules specified for the generic state variable will apply to this new class, we define `PosAndHeadSVCid` to be a subsort of `SVCid`.

```
mod POSANDHEAD-STATE-VARIABLE is
  inc POSANDHEAD-STATE-VALUE .
  inc STATE-VARIABLE .

  sort PosAndHeadSVCid .
  subsort PosAndHeadSVCid < SVCid .
endm
```

The specifics about position and heading state values imported in the above module are defined as follows.

```
fmod POSANDHEAD-STATE-VALUE is
```

```
  inc GRID .
  inc STATE-VALUE .

  sort PosAndHeadStateValue .
  subsort PosAndHeadStateValue < StateValue .

  op  _`,_dir_ : Nat Nat Dir  -> PosAndHeadStateValue .

  op _@x : PosAndHeadStateValue -> Nat .
  op _@y : PosAndHeadStateValue -> Nat .
  op _@d : PosAndHeadStateValue -> Dir .

  vars x y : Nat .
  var d : Dir .

  eq (x,y dir d)@x = x .
  eq (x,y dir d)@y = y .
  eq (x,y dir d)@d = d .
endfm
```

The constructor for the position and heading state value takes three parameters: two natural numbers indicating the location, and a direction. Operators are defined to extract the pieces of information from a state value.

3.3 Refined Interfaces in the Rover System

Some of the generic interfaces of the MDS system need to be further refined in the context of our example in order to specify the data to be communicated. This applies to the following, lower-level interfaces: (a) controller-actuator, (b) device-actuator, (c) device-sensor, and (d) estimator-sensor.

The controller will issue certain commands to the actuator. In our examples we allow two commands: turn and drive.

```
fmod PHCONTROLLER-PHACTUATOR-INTERFACE is
  inc CONTROLLER-ACTUATOR-INTERFACE .
```

```
  ops drive turn : -> Command .
endfm
```

The actuator executes these commands through communication with the device. We chose the same labels for message bodies of commands. Thus, interface between the rover and the position and heading controller looks as follows.

```
fmod ROVER-PHACTUATOR-INTERFACE is
  inc DEVICE-ACTUATOR-INTERFACE .

  ops drive turn : -> MsgBody .
endfm
```

The device, in our case the rover, will send new sensor values to the sensor. We have to define the format of those values. We decided for the following format of sensor values.

```
fmod ROVER-PHSENSOR-INTERFACE is
  inc DEVICE-SENSOR-INTERFACE .
  inc GRID .

  op posAndHead : Loc Dir -> SensorValue .
endfm
```

The sensor constructs a measurement from the sensor values. For simplicity, we chose the format for position and heading measurements to be equal to position and heading state values.

```
fmod PHESTIMATOR-PHSENSOR-INTERFACE is
  inc ESTIMATOR-SENSOR-INTERFACE .
  inc GRID .

  sort PosAndHeadMeasurement .
  subsort PosAndHeadMeasurement < Measurement .
```

```
  op _`,_dir_ : Nat Nat Dir  -> PosAndHeadMeasurement .
endfm
```

This concludes the application-specific datatypes and interfaces. Next we refine the remaining generic MDS components for the purpose of the rover example.

3.4 The Position and Heading Controller

First we need to specify what a constraint for the rover looks like. We define a sort `PosAndHeadConstraint` for this purpose. Such a constraint is the juxtaposition of two natural numbers (the intended location to be understood as $(x.y)$) and a direction. Operations on the constructor are defined to access each part of the constraint.

```
fmod POSANDHEAD-CONSTRAINT is
  pr NAT .
  pr GRID .
  inc CONSTRAINT .

  sort PosAndHeadConstraint .
  subsort PosAndHeadConstraint < Constraint .

  op ___ : Nat Nat Dir -> PosAndHeadConstraint .
           *** for now: constraint is new location and direction
           *** subsequent versions will allow for delta-x,
           *** delta-y, delta-theta, max velocity, etc.

  op _@x : PosAndHeadConstraint -> Nat .
  op _@y : PosAndHeadConstraint -> Nat .
  op _@d : PosAndHeadConstraint -> Dir .

  vars x y : Nat .
  var d : Dir .
```

```
  eq (x y d)@x = x .
  eq (x y d)@y = y .
  eq (x y d)@d = d .
endfm
```

The position and heading controller module defines the algorithm for determining the course of action to achieve a given constraint as well as the function to determine whether a particular rover state fulfills the constraint. We start with the latter. The constraint is fulfilled when all three parameters, x-, y-position and direction are achieved.

```
mod POSANDHEAD-CONTROLLER is
  pr CONTROLLER .
  inc POSANDHEAD-CONSTRAINT .
  inc POSANDHEAD-STATE-VARIABLE .
  inc PHCONTROLLER-PHACTUATOR-INTERFACE .

  sort PosAndHeadCtrlCid .
  subsort PosAndHeadCtrlCid < CtrlCid .

  *** Specification of ‘satisfy’
  var phv :   PosAndHeadStateValue .
  var phcstr : PosAndHeadConstraint .

  eq satisfy(phv,phcstr) =
     if (phv)@x == (phcstr)@x and (phv)@y == (phcstr)@y
         and (phv)@d == (phcstr)@d
     then true
     else false
     fi .
```

For the purpose of illustration we implemented a simple strategy to determine which commands should be issued to the rover in order to move towards the desired location. The strategy has three steps.

1. First the x-position of the final location is achieved. This is done by determining in which direction (east or west) along the x-axis the rover

has to drive and whether the rover needs to turn to reach its initial position. Then the number of steps is computed that the rover needs to proceed along the x-axis. An appropriate list of turn and drive commands is generated.

2. Second the final y-position of the goal is achieved. Similarly to the next step, the controller determines whether the rover needs to proceed along the y-axis in south or north direction, issue the required turning commands and then determines the number of steps for the rover.

3. Finally, the controller determines how many more turns need to be issued to get the rover in the final position.

This is a simple-minded strategy that does not take the blocked positions of the grid into account. Therefore, it is possible that the controller successfully issues all the command and still the rover did not end up in the desired position, because a drive command resulted in no change of position because of a block.

```
*** Auxiliary functions to determine the COA
op turnFromTo : Dir Dir -> Command .
op turnFor : Nat -> Command .
op driveFor : Nat -> Command .
op cmdList :  Command -> CommandList .

eq coa(phv,phcstr) =
   if (phv)@x < (phcstr)@x
   then (if (phv)@y == (phcstr)@y
         then (cmdList(turnFromTo((phv)@d, E)) ;
               cmdList(driveFor(sd((phcstr)@x,(phv)@x))) ;
               cmdList(turnFromTo(E, (phcstr)@d)))
         else (if (phv)@y < (phcstr)@y
               then (cmdList(turnFromTo((phv)@d, E)) ;
                     cmdList(driveFor(sd((phcstr)@x,(phv)@x))) ;
                     cmdList(turnFromTo(E, N)) ;
                     cmdList(driveFor(sd((phcstr)@y,(phv)@y))) ;
                     cmdList(turnFromTo(N, (phcstr)@d)) )
               else (cmdList(turnFromTo((phv)@d, E)) ;
```

```
                cmdList(driveFor(sd((phcstr)@x,(phv)@x))) ;
                cmdList(turnFromTo(E, S)) ;
                cmdList(driveFor(sd((phv)@y,(phcstr)@y))) ;
        cmdList(turnFromTo(S, (phcstr)@d)))
         fi)
      fi)
     *** use 'sd' since _-_ not defined for Nat
     *** _-_ also ambiguous with __
else (if (phv)@y == (phcstr)@y
      then (cmdList(turnFromTo((phv)@d, W)) ;
            cmdList(driveFor(sd((phcstr)@x,(phv)@x))) ;
            cmdList(turnFromTo(W, (phcstr)@d)))
      else (if (phv)@y < (phcstr)@y
           then (cmdList(turnFromTo((phv)@d, W)) ;
                 cmdList(driveFor(sd((phcstr)@x,(phv)@x))) ;
                 cmdList(turnFromTo(W, N)) ;
                 cmdList(driveFor(sd((phcstr)@y,(phv)@y))) ;
                 cmdList(turnFromTo(N, (phcstr)@d)))
           else (cmdList(turnFromTo((phv)@d, W)) ;
                 cmdList(driveFor(sd((phcstr)@x,(phv)@x))) ;
                 cmdList(turnFromTo(W, S)) ;
                 cmdList(driveFor(sd((phv)@y,(phcstr)@y))) ;
         cmdList(turnFromTo(S, (phcstr)@d)))
           fi)
     fi)
fi .

*** driving for n steps yields in a command list
*** with n "drive" commands
var n : Nat .
eq cmdList(driveFor(0)) = nil .
eq cmdList(driveFor(n)) = (drive ; cmdList(driveFor(sd(n,1)))) .

*** since the rover turns in 45 degrees, clockwise,
*** turning from one direction to another is a sequence
*** of 45 degree turns in clockwise direction
*** dir-diff computes the number of 45 degree turns necessary
var d d' : Dir .
```

```
op dir-diff : Dir Dir -> Nat .
eq dir-diff(E,E) = 0 .
eq dir-diff(E,S) = 2 .
eq dir-diff(E,W) = 4 .
eq dir-diff(E,N) = 6 .
eq dir-diff(S,S) = 0 .
eq dir-diff(S,W) = 2 .
eq dir-diff(S,N) = 4 .
eq dir-diff(S,E) = 6 .
eq dir-diff(W,W) = 0 .
eq dir-diff(W,N) = 2 .
eq dir-diff(W,E) = 4 .
eq dir-diff(W,S) = 6 .
eq dir-diff(N,N) = 0 .
eq dir-diff(N,E) = 2 .
eq dir-diff(N,S) = 4 .
eq dir-diff(N,W) = 6 .

op turnFor : Nat -> Command .
eq cmdList(turnFromTo(d,d'))
   = cmdList(turnFor(dir-diff(d,d'))) .
eq cmdList(turnFor(0)) = nil .
eq cmdList(turnFor(n))
   = (turn ; cmdList(turnFor(sd(n,1)))) .
eq cmdList(turn ; turn ; turn ; turn ; turn ;
           turn ; turn ; turn) = nil .
```

3.5 The Position and Heading Actuator

The specification of the position and heading actuator is simple in our case. The actuator takes the command from the controller and transforms it into the correct format for execution in the device. We decided to choose the same names `turn` and `drive` for commands issued to the actuator and messages sent to the rover. Thus, the specification of the position and heading actuator only needs to transform the request for a command into the appropriate rover command. Additionally, we specify the sort for the identifier of the position

and heading class and subclass it to the Actuator class identity.

```
fmod POSANDHEAD-ACTUATOR is
  inc ACTUATOR .
  inc PHCONTROLLER-PHACTUATOR-INTERFACE .
  inc ROVER-PHACTUATOR-INTERFACE .

  sort PosAndHeadActuatorCid .
  subsort PosAndHeadActuatorCid < ActuatorCid .

  *** Specification of ‘executeCmd’
  eq executeCmd(drive) = drive .
  eq executeCmd(turn) = turn .
endfm
```

3.6 The Rover

The rover specification contains all the application specific aspects. It is divided into the structural part, which defines rover attributes, and the behavioral part, which defines rules about the rover movements.

The rover knows about the grid it operates on (`grid` attribute specifying the dimension of the grid in terms of height, width and the set of blocked locations), its own position (`loc`) and heading (`hd`). The heading is given as a natural number that correspond to the degrees clockwise from the grid's north. Moreover, the rover knows whether it is currently driving, turning or in an idle state (`st` attribute).

```
mod ROVER is
  inc DEVICE .
  inc PHCONTROLLER-PHACTUATOR-INTERFACE .
  inc ROVER-PHACTUATOR-INTERFACE .
  inc ROVER-PHSENSOR-INTERFACE .
  inc GRID .

  sort RoverCid .
```

```
subsort RoverCid < DeviceCid .

*** Attributes
op grid : Nat Nat LocSet -> Attribute .
op pos_  : Loc -> Attribute .
op hd : Nat -> Attribute .
sort Status .
ops driving turning idle : -> Status .
op st :  Status -> Attribute .
```

The rover can receive a driving command from its actuator. If the rover is not heading in either north, east, south, or west direction, it will not be able to move along the grid. In this case, it will simply report back to the sensor its current position and heading, which will then be reported to the state variable. Currently, since the rover only turns in 45 degree steps, there may be situations in which the rover cannot be turned into an initial position that allows it to move along the x- or y-axes of the grid. This is not the only way that might a goal to fail. Even if the rover is directed in one of the compass directions, it still might be prevented from moving by a blocked location. Thus, the overall controller strategy will not succeed. This is due to the simplicity of our strategy and could be addressed by a more sophisticated one. Alternatively, higher-level goal achievers could learn about blocked grit positions and request moves to intermediate points that the controller can achieve.

```
vars r a s : Oid .
vars x y x' y' ht wd h : Nat .
var blocked : LocSet .
var rcid : RoverCid .

rl[drive]:
< r : rcid | myactuator(a),
             mysensor(s),
             pos (x,y),
             hd(h),
             st(idle),
             grid(ht,wd,blocked),
```

```
             ratts:AttributeSet >
msg(r,a,drive)
=>
( if (h rem 90) == 0
  then < r : rcid | myactuator(a),
                    mysensor(s),
                    pos (x,y),
                    hd(h),
                    st(driving),
                    grid(ht,wd,blocked),
                    ratts:AttributeSet >
  else < r : rcid | myactuator(a),
                    mysensor(s),
                    pos (x,y),
                    hd(h),
                    st(idle),
                    grid(ht,wd,blocked),
                    ratts:AttributeSet >
       msg(s,r,sensorValues(posAndHead((x,y),dir(h))))
 fi ) .

crl[driving]:
< r : rcid | mysensor(s),
             pos (x,y),
             hd(h),
             st(driving),
             grid(ht,wd,blocked),
             ratts:AttributeSet >
=>
< r : rcid | mysensor(s),
             pos (x',y'),
             hd(h),
             st(idle),
             grid(ht,wd,blocked),
             ratts:AttributeSet >
msg(s,r,sensorValues(posAndHead((x',y'),dir(h))))
if  (x',y') := newLoc(ht,wd,(x,y), dir(h), blocked) .
```

The operation `newLoc` is defined in the module `GRID` (see Section 3.1).

The rules for turning are very similar to the ones for driving, with the only difference that the is always able to turn.

```
rl[turn]:
< r : rcid | myactuator(a),
             pos (x,y),
             hd(h),
             st(idle),
             grid(ht,wd,blocked),
             ratts:AttributeSet >
msg(r,a,turn)
=>
< r : rcid | myactuator(a),
             pos (x,y),
             hd(h),
             st(turning),
             grid(ht,wd,blocked),
             ratts:AttributeSet >  .

rl[turning]:
< r : rcid | mysensor(s),
             pos (x,y),
             hd(h),
             st(turning),
             grid(ht,wd,blocked),
             ratts:AttributeSet >
=>
< r : rcid | mysensor(s),
             pos (x,y),
             hd((h + 45) rem 360),
             st(idle),
             grid(ht,wd,blocked),
             ratts:AttributeSet >
msg(s,r,sensorValues(posAndHead((x,y),dir(((h + 45) rem 360))))) .
```

3.7 The Position and Heading Sensor

The specification for the position and heading sensor is very simple, since the only task the sensor has in our simplistic case study is to convert sensor values into measurements. Forwarding the measurements to the estimator is part of the generic behavior of sensors and does not need further refinement.

```
fmod POSANDHEAD-SENSOR is
  inc SENSOR .
  inc DEVICE-SENSOR-INTERFACE .
  inc PHESTIMATOR-PHSENSOR-INTERFACE .
  inc ROVER-PHSENSOR-INTERFACE .

  sort PosAndHeadSensorCid .
  subsort PosAndHeadSensorCid < SensorCid .

  *** Specification of createMeasurement(sensorValue)
  var sval : SensorValue .
  vars x, y : Nat .

  eq createMeasurement(posAndHead((x,y),d:Dir)) =  ((x,y dir (d:Dir))) .
endfm
```

3.8 The Position and Heading Estimator

The position and heading estimator transforms the measurement received from the sensor into a position and heading state value.

```
mod POSANDHEAD-ESTIMATOR is
  inc ESTIMATOR .
  inc STATE-VARIABLE-ESTIMATOR-INTERFACE .
  inc PHESTIMATOR-PHSENSOR-INTERFACE .
  inc POSANDHEAD-STATE-VALUE .

  sort PosAndHeadEstCid .
```

```
  subsort PosAndHeadEstCid < EstimatorCid .

  *** Specification of ‘verifyState(measurement)’
  vars x, y : Nat .
  eq verifyState((x,y dir (d:Dir))) = (x,y dir (d:Dir)) .
endm
```

This concludes the specification of the components of the rover system. In the next chapter we will address how the system is composed and instantiated for purposes of analysis.

Chapter 4

Analysis of the Rover System

4.1 An Example Scenario

To analyze our small system, we set up a scenario, consisting of a rover and its associated MDS components (actuator, sensor, etc.). Moreover, we define a specific grid on which the rover will move. The grid has 3 blocked locations.

```
fmod GRID-TEST is
  inc GRID .
  op B : -> LocSet .
  eq B = (1,1) (2,2) (3,3) .
endfm
```

The following module defines a class `Rover` and an instance of that class, named `MyRover`. The size of our test grid is 5×5 and has the three blocked locations defined in `GRID-TEST`. The initial position of the rover is at location $(0, 0)$ and the rover is headed towards east.

```
mod ROVER-TEST is
  inc GRID-TEST .
  inc ROVER .

  op rov : -> Object .
```

```
  op Rover : -> RoverCid .

  eq rov = < o("MyRover") : Rover |
                myactuator(o("MyPosAndHeadActuator")),
                mysensor(o("MyPosAndHeadSensor")),
                pos(0,0),
                hd(90),
                st(idle),
                grid(5,5,B) > .
endm
```

As mentioned earlier, the centerpiece of a state-based system specification is the state variable. For experimentation purposes, we instantiate the position and heading state variable class in the following way:

```
mod POSANDHEAD-STATE-VARIABLE-TEST is
  inc POSANDHEAD-STATE-VARIABLE .

  op PosAndHeadStateVar : -> PosAndHeadSVCid .

  op myphsv : -> Object .
  eq myphsv =
     < o("MyPosAndHeadStateVar") : PosAndHeadStateVar |
               myctrl(o("MyPosAndHeadCtrl")),
               myest(o("MyPosAndHeadEstimator")),
               req(o("NoOid")),
               cstr(noCstr),
               val((0,0 dir(E))) > .
endm
```

The state variable reflects correctly the initial position and heading of the rover. It has references to its controller and estimator objects. The initial values for a requestor and a constraint are set to default values.

Similarly, all other components of the rover system are instantiated. The most crucial thing about setting up the instances of the classes is to intialize attributes such that rules are applicable. For example, the rule labeled `readyForNewConstraint` for controllers requires some values for the

attributes `mysv` and `waitAfter`. We connect the rover system components with each other by setting appropriate values for references to other objects, such as `mysv(o("MyPosAndHeadStateVar"))`, and by assigning initial values to other attributes (such as `waitAfter(noMsg)`).

```
mod POSANDHEAD-CONTROLLER-TEST is
  inc POSANDHEAD-CONTROLLER .

  op PosAndHeadCtrl : -> PosAndHeadCtrlCid .

  op noOid : -> Oid .
  op myphctrl : -> Object .
  eq myphctrl =
     < o("MyPosAndHeadCtrl") : PosAndHeadCtrl |
                 mysv( o("MyPosAndHeadStateVar")) ,
                 myactuator( o("MyPosAndHeadActuator")) ,
                 currentCstr(noCstr) ,
                 currentCstrReq(noOid) ,
                 currentSVVal(uk) ,
                 cmds(noCmd) ,
                 waitAfter(noMsg) > .
endm

mod POSANDHEAD-ACTUATOR-TEST is
  inc POSANDHEAD-ACTUATOR .

  op PosAndHeadActuator : -> PosAndHeadActuatorCid .

  op myphactuator : -> Object .
  eq myphactuator =
     < o("MyPosAndHeadActuator") : PosAndHeadActuator |
                            mydevice(o("MyRover")),
                            myctrl(o("MyPosAndHeadCtrl")),
                            executedCmd(noCmd) > .
endm

mod POSANDHEAD-SENSOR-TEST is
  inc POSANDHEAD-SENSOR .
```

```
  op PosAndHeadSensor : -> PosAndHeadSensorCid .

  op myphsensor : -> Object .
  eq myphsensor =
     < o("MyPosAndHeadSensor") : PosAndHeadSensor |
                       mydevice(o("MyRover")),
                       myest(o("MyPosAndHeadEstimator")),
                       measurement(noMeas) > .
endm

mod POSANDHEAD-ESTIMATOR-TEST is
  inc POSANDHEAD-ESTIMATOR .

  op PosAndHeadEst : -> PosAndHeadEstCid .

  op myphest : -> Object .
  eq myphest =
     < o("MyPosAndHeadEstimator") : PosAndHeadEst |
                       mysensor(o("MyPosAndHeadSensor")),
                       mysv(o("MyPosAndHeadStateVar")),
                       state(uk),
                       waitAfter(noMsg) > .
endm
```

The SYSTEM module combines the component test modules, including all necessary generic components, the application specific components, and their instantiations. Analyses will be done in the context of this module.

```
mod SYSTEM is
  inc GRID .
  inc POSANDHEAD-STATE-VARIABLE-TEST .
  inc POSANDHEAD-CONTROLLER-TEST .
  inc POSANDHEAD-ACTUATOR-TEST .
  inc POSANDHEAD-SENSOR-TEST .
  inc POSANDHEAD-ESTIMATOR-TEST .
  inc ROVER-TEST .
endm
```

4.2 Some Example Runs

4.2.1 Using Maude to run the Rover Example

We assume the reader has installed the Maude system (instructions can be found on the Maude website (`maude.cs.uiuc.edu`), that the command `maude` is aliased to the maude executable, and that MAUDE_LIB environment variable is set to give the location of the files model-checker.maude and prelude.maude. Further assume that the file `rover.m` contains commands to load files containing the MDS and Rover modules. (Note that the order in which the modules are loaded matters. Modules have to be defined before they can be referenced in another module.)

Start Maude in the directory containing `rover.m` using the `maude` command. This will result in a banner such as the following to be printed.

```
                     \||||||||||||||||||/
                   --- Welcome to Maude ---
                     /||||||||||||||||||\
        Maude alpha 80a built: May 23 2003 21:08:07
             Copyright 1997-2003 SRI International
                   Wed Jul 23 21:32:09 2003
Maude>
```

Now the rover system is loaded into Maude by

```
Maude> in rover.m
```

As a result the Maude interpreter prints the following list of modules defined.

```
==========================================
mod MYCONF
==========================================
fmod ATTRIBUTES
==========================================
fmod CONSTRAINT
```

```
============================================
fmod REASON
============================================
fmod STATE-VARIABLE-ENVIRONMENT-INTERFACE
============================================
fmod STATE-VALUE
============================================
fmod STATE-VARIABLE-CONTROLLER-INTERFACE
============================================
fmod STATE-VARIABLE-ESTIMATOR-INTERFACE
============================================
fmod COMMAND
============================================
fmod CONTROLLER-ACTUATOR-INTERFACE
============================================
fmod DEVICE-ACTUATOR-INTERFACE
============================================
fmod SENSOR-VALUE
============================================
fmod DEVICE-SENSOR-INTERFACE
============================================
fmod MEASUREMENT
============================================
fmod ESTIMATOR-SENSOR-INTERFACE
============================================
mod STATE-VARIABLE
============================================
mod ACTUATOR
============================================
mod ACTUATOR-TEST
============================================
mod SENSOR
============================================
mod SENSOR-TEST
============================================
mod DEVICE
============================================
mod DEVICE-TEST
```

```
============================================
fmod COMMAND-LIST
============================================
mod CONTROLLER
============================================
mod CONTROLLER-TEST
============================================
mod ESTIMATOR
============================================
mod ESTIMATOR-TEST
============================================
fmod GRID
============================================
fmod GRID-TEST
============================================
fmod PHCONTROLLER-PHACTUATOR-INTERFACE
============================================
fmod ROVER-PHACTUATOR-INTERFACE
============================================
fmod ROVER-PHSENSOR-INTERFACE
============================================
fmod PHESTIMATOR-PHSENSOR-INTERFACE
============================================
fmod POSANDHEAD-STATE-VALUE
============================================
mod POSANDHEAD-STATE-VARIABLE
============================================
mod POSANDHEAD-STATE-VARIABLE-TEST
============================================
fmod POSANDHEAD-CONSTRAINT
============================================
mod POSANDHEAD-CONTROLLER
============================================
mod POSANDHEAD-CONTROLLER-TEST
============================================
fmod POSANDHEAD-ACTUATOR
============================================
mod POSANDHEAD-ACTUATOR-TEST
```

```
==========================================
fmod POSANDHEAD-SENSOR
==========================================
mod POSANDHEAD-SENSOR-TEST
==========================================
mod ROVER
==========================================
mod ROVER-TEST
==========================================
mod POSANDHEAD-ESTIMATOR
==========================================
mod POSANDHEAD-ESTIMATOR-TEST
==========================================
mod SYSTEM
Maude>
```

4.2.2 Default Execution of the Rover Scenario

After loading the system, one can perform various tests. The first step is to execute the system for a given initial configuration. A default strategy will be chosen by the interpreter. We chose as an initial system configuration the rover plus the appropriate instances of the other five core classes (statevariable, controller, estimator, sensor, and actuator). Moreover, we add a message addressed to the position and heading state variable with a request for a new constraint. The goal is to move the rover from its original location $(0,0)$ heading east to the new location $(1,0)$ also heading east. This configuration is executed using the `rew` (short for rewrite) command.

```
Maude> rew myphctrl myphsv myphactuator myphsensor rov myphest
    msg(o("MyPosAndHeadStateVar"), o("MyRequester"),
        startConstraint((1 0 E))) .
```

The execution terminates with the following final configuration.

```
rewrite in SYSTEM :
    myphctrl myphsv myphactuator myphsensor rov myphest
```

```
    msg(o("MyPosAndHeadStateVar"), o("MyRequester"),
        startConstraint(1 0 E)) .
rewrites: 89 in 10ms cpu (5ms real) (8900 rewrites/second)
result Configuration:
< o("MyPosAndHeadActuator") : PosAndHeadActuator |
    myctrl(o("MyPosAndHeadCtrl")),
    mydevice(o("MyRover")),
    executedCmd(drive) >
< o("MyPosAndHeadCtrl") : PosAndHeadCtrl |
    mysv(o("MyPosAndHeadStateVar")),
    myactuator(o("MyPosAndHeadActuator")),
    waitAfter(noMsg),currentCstr(1 0 E),
    currentCstrReq(o("MyRequester")),
    currentSVVal(1,0 dir E),cmds(nil) >
< o("MyPosAndHeadEstimator") : PosAndHeadEst |
    mysv(o("MyPosAndHeadStateVar")),
    mysensor(o("MyPosAndHeadSensor")),
    waitAfter(noMsg),
    state(1,0 dir E) >
< o("MyPosAndHeadSensor") : PosAndHeadSensor |
    myest(o("MyPosAndHeadEstimator")),
    mydevice(o("MyRover")),
    measurement(1,0 dir E) >
< o("MyPosAndHeadStateVar") : PosAndHeadStateVar |
     myest(o("MyPosAndHeadEstimator")),
     myctrl(o("MyPosAndHeadCtrl")),
     val(1,0 dir E),
     req(o("MyRequester")),
     cstr(1 0 E) >
< o("MyRover") : Rover |
     myactuator(o("MyPosAndHeadActuator")),
     mysensor(o("MyPosAndHeadSensor")),
     pos (1,0),
     hd(90),
     st(idle),
     grid(5, 5, (1,1) (2,2) (3,3)) >
msg(o("MyRequester"), o("MyPosAndHeadStateVar"),
    constraintSuccess(1 0 E))
```

The rover has changed its position to the desired location and also satisfies the direction requirements. A message is sent back to the requester of the goal, announcing that the goal was sucessfully achieved. The other components (actuator, controller, etc.) still hold some of the values that were used during processing the constraint.

An example in which the constraint cannot be fulfilled is a constraint where the rover is asked to move to location $(1, 2)$. Though this is not a blocked location, the simplistic strategy of the controller is not issuing commands that will maneuver around the blocked locations. Thus, rewriting the initial configuration with a goal to move to $(1, 2)$

```
Maude> rew  myphctrl myphsv myphactuator myphsensor rov myphest
    msg(o("MyPosAndHeadStateVar"), o("MyRequester"),
        startConstraint((1 2 E))) .
```

ends up in a configuration where the requester is told that the goal was not achieved by the course of action determined by the controller.

```
Maude> ...
msg(o("MyRequester"), o("MyPosAndHeadStateVar"),
    constraintFailure(1 2 E, COANoSuccess))
```

If one wants to understand the steps taken in the default rewrite, one can start with a one step rewrite (i.e., default application of one rewrite rule)

```
Maude> rew [1] myphctrl myphsv myphactuator ... .
```

and continuing each step with

```
Maude> continue 1 .
```

4.2.3 Search, Model-checking and Other Options

Built-in search strategies in Maude allow more sophisticated analyses. If one wants to find all possible configurations from a given initial configuration, one can execute the search command (filling in the dots with the initial configuration).

```
Maude> search ... =>! c:Configuration .
```

Applying this search strategy to our initial configuration resulted in exactly one solution, namely the configuration that is the result of the default execution. The Maude output reads as follows.

```
Maude> search in SYSTEM : ...
Solution 1 (state 15)
states: 16  rewrites: 109 in 10ms cpu (9ms real) (10900 rewrites/second)
c:Configuration --> ...
  msg(o("MyRequester"), o("MyPosAndHeadStateVar"),
      constraintSuccess(1 0 E))

No more solutions.
```

The search indicates that the system underwent 16 state changes due to applications of rewrite rules before terminating in the given configuration. To see which are these states and what rules have been applied, one can ask Maude to show the search graph.

```
Maude> show search graph .
```

Showing the search graph for our example results in a series of 16 states (numbered from 0 to 15). For each state the configuration is given, the rule that has been applied and the label of the rule.

```
Maude> show search graph .
state 0, Configuration: ...
arc 0 ===> state 1 (rl ... [label startConstraintReadyReq] .)
```

4.3 Lessons Learned from First Analyses

Before we focus on more sophisticated testing procedures, we briefly summarize what errors had been found by just using Maude as an executable specification method.

Some minor problems were already discovered during the specification process. Syntactical errors are found by the parser. After the specification parsed correctly, the default execution of the initial configuration did not terminate in earlier versions of the rover system, due to incorrect specification of the controller strategy and some specification mistakes in other modules (such as equations for assigning directions to numbers in the grid).

Another typical error is that initialization of class instances is incomplete, resulting in the inapplicability of rules. Thus, rewriting an initial scenario that *should* result in a specific configuration, just did not behave as expected. Since not all rules could be applied, the final configuration was not the expected one.

Other specification errors caused the infinite loops. Rewriting the initial configuration step-by-step uncovered the rules that were not behaving as expected. Most often this was due to the enabling conditions on the rule (the lefthand side configuration template). We had to tighten the applicability of rules by either requiring an attribute to have a certain value or by introducing in various places the `waitAfter` attribute and preventing concurrent behavior inside a component.

This round of specification, analysis, correction, yielded the specification of the rover system as presented in the previous chapter. At this point, we had gained some confidence that the rover system represents to a first approximation the model that we had in mind. Nevertheless, further testing brought more problems to the light which will be addressed in the following section.

The next series of tests was aimed in testing the stability of the system in the face of concurrent goal requests. We changed our initial configuration by adding a second constraint request.

```
search myphctrl myphsv myphactuator myphsensor rov myphest
       msg(o("MyPosAndHeadStateVar"), o("MyRequester"),
           startConstraint((2 0 E)))
       msg(o("MyPosAndHeadStateVar"), o("MyRequester"),
           startConstraint((1 0 E)))
=>! c:Configuration .
```

The two anticipated outcomes of rewriting this system are that the rover

system either

1. completes processing one goal, reporting back to the requester and then processes the other goal (both goals are achievable if processed sequentially in either order), or

2. processes one goal and while in the process of trying to achieve the first goal, receives the request for the other goal and reports back that it is not achievable (since the controller is not ready for a new goal).

Given the symmetry in both alternatives, we expected four solutions when searching for all possible configurations. We were surprised to get 16 solutions.

Investigating the cause of the 16 solutions we found the following two problems.

The state variable accepts a new constraint while another constraint is still being processed. As a result the attribute that records the current constraint is overwritten and lost. Here the original rule:

```
rl[startConstraintReadyReq]:
  < sv : svcid | myctrl(ctrl),
                 req(o'),
                 cstr(cstr'),
                 svatts:AttributeSet >
  msg(sv,o,startConstraint(cstr))
  =>
  < sv : svcid | myctrl(ctrl),
                 req(o),
                 cstr(cstr),
                 svatts:AttributeSet >
  msg(ctrl,sv,readyReq)  .
```

The new rule only allows to process a new constraint when no other constraint is currently processed. This is implemented using the `waitAfter` attribute.

```
  rl[startConstraintReadyReq]:
```

```
< sv : svcid | myctrl(ctrl),
               req(o'),
               cstr(cstr'),
               waitAfter(noMsg),
               svatts:AttributeSet >
msg(sv,o,startConstraint(cstr))
=>
< sv : svcid | myctrl(ctrl),
               req(o),
               cstr(cstr),
               waitAfter(msg(sv,o,startConstraint(cstr))),
               svatts:AttributeSet >
msg(ctrl,sv,readyReq) .
```

Similarly, the controller replies that it is ready for a new constraint if it is currently not issuing commands to process a constraint, but the controller does not remember that it has executed this rule. Therefore, the controller can positively reply to several constraints with respect to its readiness, where it only should be replying to one constraint and then be blocked until the constraint was received and processed. We accounted for this in the `waitAfter` attribute of the if-case on the righthand side of the following new rule:

```
rl[readyForNewConstraint]:
< c : ccid | mysv(sv),
             waitAfter(msg),
             ctrlatts:AttributeSet >
msg(c,sv,readyReq)
=>
if (msg == noMsg)
then < c : ccid | mysv(sv),
                  waitAfter(msg(c,sv,readyReq)),
                  ctrlatts:AttributeSet >
     msg(sv,c,readyRep(true))
else < c : ccid | mysv(sv),
                  waitAfter(msg),
                  ctrlatts:AttributeSet >
     msg(sv,c,readyRep(false))
fi .
```

Consequently, the lefthand side of the subsequent rule is changed to (again, change does only apply to `waitAfter` attribute):

```
rl[getCstrCompareGoalIssueCommand]:
< c : ccid | mysv(sv),
             myactuator(a),
             currentCstr(cstr'),
             currentCstrReq(o'),
             currentSVVal(v'),
             cmds(cl),
             waitAfter(msg(c,sv,readyReq)),
             ctrlatts:AttributeSet >
msg(c,sv,startCstr(cstr,o,v))
=> ...
```

Bibliography

[BGH+03a] J. Bhuta, E. Gradman, L. Huang, A. Lam, K. Lee, S. Madrigal, S. Meyers, and G. Perez. Inspector SCRover (ISCR): State analysis, version 1.0, 2003. http://matador.usc.edu:8888/export_package/wall-following_scenario/spec%ification_documents.

[BGH+03b] J. Bhuta, E. Gradman, L. Huang, A. Lam, K. Lee, S. Madrigal, S. Meyers, and G. Perez. Inspector SCRover (ISCR): System and software architecture description (SSAD), version 1.4, 2003. http://matador.usc.edu:8888/export_package/wall-following_scenario/spec%ification_documents.

[BGH+03c] J. Bhuta, E. Gradman, L. Huang, A. Lam, K. Lee, S. Madrigal, S. Meyers, and G. Perez. Inspector SCRover (ISCR): System and software requirements definition (SSRD), version 1.2, 2003. http://matador.usc.edu:8888/export_package/wall-following_scenario/spec%ification_documents.

[CDE+99] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Marti-Oliet, J. Meseguer, and J. Quesada. Maude: Specification and programming in rewriting logic, 1999. http://maude.csl.sri.com/manual.

[CELM96] M. Clavel, S. Eker, P. Lincoln, and J. Meseguer. Principles of maude. In Meseguer [Mes96], pages 65–89.

[CM96] M. Clavel and J. Meseguer. Reflection and strategies in rewriting logic. In Meseguer [Mes96], pages 125–147.

[DRRS00] D. Dvorak, R. Rasmussen, G. Reeves, and A. Sacks. Software Architecture Themes In JPL's Mission Data System. In *IEEE Aerospace Conference, USA*, 2000. http://techreports.jpl.nasa.gov/1999/99-1886.pdf.

[Mes92] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[Mes96] J. Meseguer, editor. *Rewriting Logic and Its Applications, First International Workshop, Asilomar Conference Center, Pacific Grove, CA, September 3-6, 1996*. Elsevier Science B.V., Electronic Notes in Theoretical Computer Science, Volume 4, http://www.elsevier.nl/locate/entcs/volume4.html, 1996.

[Mes00] J. Meseguer. Rewriting logic and Maude: A wide-sprectrum semantic framework for object-based distributed systems. In S.F. Smith and C.L. Talcott, editors, *Proc. IFIP Conf. on Formal Methods of Open Object-Based Distributed Systems VI (FMOODS 2000), September 6-8, 2000, Stanford, CA*, pages 89–117. Kluwer Academic Publishers, 2000.

[MPPW98] N. Muscetolla, P. Pandurang, B. Pell, and B. Williams. Remote Agent: To Boldly Go Where No AI System Has Gone Before. *Artificial Intelligence*, 103(1-2):5–48, August 1998. http://ic.arc.nasa.gov/projects/Executive/papers/aij98.pdf.

[PBC+98] B. Pell, D.E. Bernard, S.A. Chien, E. Gat, N. Muscetolla, P. Pandurang, M.D. Wagner, and B. Williams. An Autonomous Spacecraft Agent Prototype. *Autonomous Robots*, 5(1), March 1998. http://ic.arc.nasa.gov/projects/Executive/papers/arj98.pdf.

Appendix A

Maude Specification

A.1 MDS Architecture

```
mod MYCONF is
  inc CONFIGURATION .
  inc STRING .

  op o : String  -> Oid .

  sort MsgBody .
  op msg : Oid Oid MsgBody -> Msg .

  op noMsg : -> Msg .
endm
```

A.1.1 Interfaces

```
*****************************************************************************
***                            Interfaces                                 ***
***                                                                       ***
*****************************************************************************

fmod ATTRIBUTES is
  inc MYCONF .
```

```
  op myest  : Oid -> Attribute .
  op mysv  : Oid -> Attribute .
  op myctrl  : Oid -> Attribute .
  op mydevice  : Oid -> Attribute .
  op myactuator : Oid -> Attribute .
  op mysensor  : Oid -> Attribute .
     *** for simplicity, we defined the above attributes in one module
     *** that is to be imported in all other module
     *** correct use
     *** (could have been reflected in more complicated module structure)
     *** sv knows ctrl and estimator (and vice a versa)
     *** ctrl knows actuator (and vice a versa)
     *** estimator knows sensor (and vice a versa)
     *** device knows actuator and sensor (and vice a versa)


  op waitAfter : Msg -> Attribute .
     *** attribute for control-flow within an object
     *** contains the last message received
     *** if there was no last message received, then it contains the current
     *** message sent
endfm

*****************************************************************************
***               StateVariable-Environment-Interface                     ***
***  messages between controller and statevariable                        ***
***                                                                       ***
*****************************************************************************

fmod CONSTRAINT is
  sort Constraint .
  op noCstr : -> Constraint .
endfm

fmod REASON is
  sort Reason .
  op noReason : -> Reason .
endfm
```

```
fmod STATE-VARIABLE-ENVIRONMENT-INTERFACE is
  inc MYCONF .
  inc CONSTRAINT .
  inc REASON .

  op startConstraint : Constraint -> MsgBody .
  op constraintSuccess : Constraint -> MsgBody .
  op constraintFailure : Constraint Reason -> MsgBody .
                         *** Reason - for failure
endfm

****************************************************************************
***               StateVariable-Controller-Interface                     ***
***  messages between controller and statevariable                       ***
***                                                                      ***
****************************************************************************


fmod STATE-VALUE is
  sort StateValue .

  sort UnknownStateValue .
  subsort UnknownStateValue < StateValue .

  op uk : -> UnknownStateValue .
endfm


fmod STATE-VARIABLE-CONTROLLER-INTERFACE is
  inc MYCONF .
  inc CONSTRAINT .
  inc REASON .
  inc STATE-VALUE .


  op readyReq : -> MsgBody .
  op readyRep : Bool -> MsgBody .

  op startCstr : Constraint Oid StateValue -> MsgBody .
  op endCstr : Constraint Oid  Reason -> MsgBody .
```

```
  op newVal : StateValue -> MsgBody .

endfm


*****************************************************************************
***              StateVariable-Estimator-Interface                       ***
***  messages between controller and statevariable                       ***
***                                                                       ***
*****************************************************************************

fmod STATE-VARIABLE-ESTIMATOR-INTERFACE is
  inc MYCONF .
  inc STATE-VALUE .

  op updStateReq : StateValue -> MsgBody .
  op ackUpdStateReq : StateValue -> MsgBody .
endfm


*****************************************************************************
***                Controller-Actuator-Interface                         ***
***                                                                       ***
*****************************************************************************

fmod COMMAND is
  sort Command .
  op noCmd : -> Command .
endfm


fmod CONTROLLER-ACTUATOR-INTERFACE is
  inc MYCONF .
  inc COMMAND .

  op issueCmd : Command -> MsgBody .
endfm

*****************************************************************************
```

```
***                    Device-Actuator-Interface                          ***
***                                                                       ***
*****************************************************************************

fmod DEVICE-ACTUATOR-INTERFACE is
  inc MYCONF .
  inc COMMAND .

  op executeCmd : Command -> MsgBody .

endfm


*****************************************************************************
***                    Device-Sensor-Interface                            ***
***                                                                       ***
*****************************************************************************

fmod SENSOR-VALUE is
  sort SensorValue .
endfm

fmod DEVICE-SENSOR-INTERFACE is
  inc MYCONF .
  inc SENSOR-VALUE .

  op sensorValues : SensorValue -> MsgBody .
endfm


*****************************************************************************
***                    Estimator-Sensor-Interface                            ***
***                                                                       ***
*****************************************************************************

fmod MEASUREMENT is
  sort Measurement .
  op  noMeas : -> Measurement .
endfm
```

```
fmod ESTIMATOR-SENSOR-INTERFACE is
  inc MYCONF .
  inc MEASUREMENT .

  op newMeasurement : Measurement -> MsgBody .
endfm
```

A.1.2 State Variable

```
****************************************************************************
***                    StateVariable                                      ***
***  has a StateVal                                                       ***
***  knows its estimator, controller                                      ***
****************************************************************************


mod STATE-VARIABLE is
  inc ATTRIBUTES .
  inc STATE-VARIABLE-ENVIRONMENT-INTERFACE .
  inc STATE-VARIABLE-CONTROLLER-INTERFACE .
  inc STATE-VARIABLE-ESTIMATOR-INTERFACE .

  sort SVCid .
  subsort SVCid < Cid .

  *** Attributes
  op  val : StateValue -> Attribute .
  op  req : Oid -> Attribute .
  op  cstr :  Constraint -> Attribute .

  *** Reasons
  op notReady : -> Reason .

  vars sv o o' ctrl est : Oid .
  var v v2 : StateValue .
  var svcid : SVCid .
  var cstr cstr' : Constraint .
  var b : Bool .
  var r : Reason .
```

```
rl[startConstraintReadyReq]:
< sv : svcid | myctrl(ctrl),
               req(o'),
               cstr(cstr'),
               waitAfter(noMsg),
               svatts:AttributeSet >
msg(sv,o,startConstraint(cstr))
=>
< sv : svcid | myctrl(ctrl),
               req(o),
               cstr(cstr),
               waitAfter(msg(sv,o,startConstraint(cstr))),
               svatts:AttributeSet >
msg(ctrl,sv,readyReq) .

rl[forwardConstraint]:
< sv : svcid | myctrl(ctrl),
               req(o),
               cstr(cstr),
               waitAfter(msg(sv,o,startConstraint(cstr))),
               val(v),
               svatts:AttributeSet >
msg(sv,ctrl,readyRep(b))
=>
if b
then < sv : svcid | myctrl(ctrl),
                    req(o),
                    cstr(cstr),
                    waitAfter(msg(sv,ctrl,readyRep(b))),
                    val(v),
                    svatts:AttributeSet >
     msg(ctrl,sv,startCstr(cstr,o,v))
     *** Optimization: SV sends current value with constraint to controller
else < sv : svcid | myctrl(ctrl),
                    req(o),
                    cstr(cstr),
                    waitAfter(noMsg),
                    val(v),
                    svatts:AttributeSet >
```

```
      msg(o,sv,constraintFailure(cstr,notReady))
  fi .

  rl[stateValUpdNotifyCtrlAckEst] :
  < sv : svcid | myctrl(ctrl),
                 myest(est),
                 val(v),
                 svatts:AttributeSet >
  msg(sv,est,updStateReq(v2))
  =>
  < sv : svcid | myctrl(ctrl),
                 myest(est),
                 val(v2),
                 svatts:AttributeSet >
  msg(ctrl,sv,newVal(v2))
  msg(est,sv,ackUpdStateReq(v2)) .

  rl[endConstraint] :
  < sv : svcid | myctrl(ctrl),
                 waitAfter(msg(sv,ctrl,readyRep(true))),
                 svatts:AttributeSet >
  msg(sv,ctrl,endCstr(cstr,o,r))
  =>
  if (r == noReason)
  then < sv : svcid | myctrl(ctrl),
                      waitAfter(noMsg),
                      svatts:AttributeSet >
       msg(o,sv,constraintSuccess(cstr))
  else < sv : svcid | myctrl(ctrl),
                      waitAfter(noMsg),
                      svatts:AttributeSet >
       msg(o,sv,constraintFailure(cstr,r))
  fi .
endm

mod STATE-VARIABLE-TEST is
  inc STATE-VARIABLE .

  op StateVar : -> SVCid .
```

```
  op mystatevar : -> Object .
  eq mystatevar = < o("MyStateVar") : StateVar |  myctrl(o("MyCtrl")),
                                                  myctrl(o("MyEstimator")),
                                                  req(o("noOid")),
                                                  cstr(noCstr),
                                                  val(uk) > .
endm
```

A.1.3 Actuator

```
****************************************************************************
***                 Actuator (HardwareAdapter)                           ***
***  knows its Device, knows its Controller                              ***
***                                                                      ***
****************************************************************************

mod ACTUATOR is
  inc ATTRIBUTES .
  inc CONTROLLER-ACTUATOR-INTERFACE .
  inc DEVICE-ACTUATOR-INTERFACE .

  sort ActuatorCid .
  subsort ActuatorCid < Cid .


  *** Attributes
  op  executedCmd : Command -> Attribute .
      *** the command that was sent by the controller for execution

  vars act d ctrl : Oid .
  var actcid  : ActuatorCid .
  vars cmd cmd' : Command .

  rl[executeCommandReq]:
  < act : actcid | mydevice(d),
                   myctrl(ctrl),
                   executedCmd(cmd'),
                   actatts:AttributeSet >
  msg(act,ctrl,issueCmd(cmd))
```

```
  =>
  < act : actcid | mydevice(d),
                   myctrl(ctrl),
                   executedCmd(cmd),
                   actatts:AttributeSet >
  msg(d,act,executeCmd(cmd)) .


endm

mod ACTUATOR-TEST is
  inc ACTUATOR .

  op Actuator : -> ActuatorCid .

  op myactuator : -> Object .
  eq myactuator =
     < o("MyActuator") : Actuator | mydevice(o("MyDevice")),
                                    myctrl(o("MyController")),
                                    executedCmd(noCmd) > .

endm
```

A.1.4 Sensor

```
****************************************************************************
***                 Sensor (HardwareAdapter)                             ***
***  knows its Device, knows its Estimator                               ***
***                                                                      ***
****************************************************************************

mod SENSOR is
  inc ATTRIBUTES .
  inc ESTIMATOR-SENSOR-INTERFACE .
  inc DEVICE-SENSOR-INTERFACE .

  sort SensorCid .
  subsort SensorCid < Cid .
```

```
  *** Attributes
  op  measurement : Measurement -> Attribute .
      *** for now only the last measurement is stored, not the whole history

  *** Converts values to measurements
  op createMeasurement : SensorValue -> Measurement .

  vars d est sens : Oid .
  var  senscid  : SensorCid .
  vars sval : SensorValue .
  var  m : Measurement .


 rl[processSensorValuesSendNewMeasurementToEstimator]:
  < sens : senscid | myest(est),
                     mydevice(d),
             measurement(m),
                     sensatts:AttributeSet >
  msg(sens,d,sensorValues(sval))
  =>
  < sens : senscid | myest(est),
                     mydevice(d),
             measurement(createMeasurement(sval)),
                     sensatts:AttributeSet >
  msg(est,sens,newMeasurement(createMeasurement(sval))) .
  *** Optimization:
  *** sensor sends ack of new values to estimator,
  *** estimator requests values, sensor replies values
endm

mod SENSOR-TEST is
  inc SENSOR .

  op Sensor : -> SensorCid .

  op mysensor : -> Object .
  eq mysensor =
     < o("MySensor") : Sensor | mydevice(o("MyDevice")),
                                myest(o("MyEstimator")),
                                measurement(noMeas) > .
```

```
endm
```

A.1.5 Device

```
**************************************************************************
***                        Device                                      ***
***                                                                    ***
***                                                                    ***
**************************************************************************

mod DEVICE is
  inc ATTRIBUTES .
  inc DEVICE-ACTUATOR-INTERFACE .
  inc DEVICE-SENSOR-INTERFACE .

  sort DeviceCid .
  subsort DeviceCid < Cid .

endm


mod DEVICE-TEST is
  inc DEVICE .

  op Device : -> DeviceCid .

  op mydevice : -> Object .

  eq mydevice =
     < o("MyDevice") : Device | mysensor(o("MySensor")),
                                myactuator(o("MyActuator")) > .
endm
```

A.1.6 Controller

```
fmod COMMAND-LIST is
  inc COMMAND .
  sort CommandList .
  subsort Command < CommandList .

  op nil : -> CommandList .
  op _;_ : CommandList CommandList -> CommandList [ctor assoc id: nil] .
  op first : CommandList -> Command .
  op rest : CommandList -> CommandList .

  var cmdl : CommandList .
  var cmd : Command .

  eq first(nil) = noCmd .
  eq first(cmd ; cmdl) = cmd .
  eq rest(nil) = noCmd .
  eq rest(cmd ; cmdl) = cmdl .
endfm

******************************************************************************
***                   Controller                                           ***
***  knows its StateVariable, Actuator                                     ***
***  saves current constraint and the requester of that constraint         ***
***  also keeps lates value of state variable                              ***
******************************************************************************

mod CONTROLLER is
  inc ATTRIBUTES .
  inc STATE-VARIABLE-CONTROLLER-INTERFACE .
  inc CONTROLLER-ACTUATOR-INTERFACE .
  inc COMMAND-LIST .

  sort CtrlCid .
  subsort CtrlCid < Cid .


  *** Attributes
```

```
op  currentCstr : Constraint -> Attribute .
                 *** constraint that is currently handled
op  currentCstrReq : Oid -> Attribute .
                 *** requester who started current constraint
op  currentSVVal : StateValue -> Attribute .
                 *** last requested state value of state variable
op cmds : CommandList -> Attribute .

*** Internal computations
op satisfy : StateValue Constraint -> Bool .
             *** tests whether current state satisfies constraint
op coa : StateValue Constraint -> CommandList .
     *** determines course of action,
             *** given current value and constraint

*** Reason for failure:
*** course of action has been executed, constraint still not satisfied
op COANoSuccess : -> Reason .


vars c a sv o o' : Oid .
var msg : Msg .
vars cstr cstr' : Constraint .
var v v' v'' : StateValue .
var b : Bool .
var r : Reason .
var ccid  : CtrlCid .
var cl : CommandList .



rl[readyForNewConstraint]:
< c : ccid | mysv(sv),
             waitAfter(msg),
             ctrlatts:AttributeSet >
msg(c,sv,readyReq)
=>
if (msg == noMsg)
then < c : ccid | mysv(sv),
                 waitAfter(msg(c,sv,readyReq)),
```

```
                  ctrlatts:AttributeSet >
     msg(sv,c,readyRep(true))
else < c : ccid | mysv(sv),
                  waitAfter(msg),
                  ctrlatts:AttributeSet >
     msg(sv,c,readyRep(false))
fi .


rl[getCstrCompareGoalIssueCommand]:
< c : ccid | mysv(sv),
             myactuator(a),
             currentCstr(cstr'),
             currentCstrReq(o'),
             currentSVVal(v'),
             cmds(cl),
             waitAfter(msg(c,sv,readyReq)),
             ctrlatts:AttributeSet >
msg(c,sv,startCstr(cstr,o,v))
=>
if satisfy(v,cstr)
then < c : ccid | mysv(sv),
                  myactuator(a),
                  currentCstr(cstr),
                  currentCstrReq(o),
                  currentSVVal(v),
                  cmds(cl),
                  waitAfter(noMsg),
                  ctrlatts:AttributeSet >
     msg(sv,c,endCstr(cstr,o,noReason))
else < c : ccid | mysv(sv),
                  myactuator(a),
                  currentCstr(cstr),
                  currentCstrReq(o),
                  currentSVVal(v),
                  cmds(rest(coa(v,cstr))),
                  waitAfter(msg(c,sv,startCstr(cstr,o,v))),
                  ctrlatts:AttributeSet >
     msg(a,c,issueCmd(first(coa(v,cstr))))
fi .
```

```
rl[getNewValueCompareGoal]:
< c : ccid | mysv(sv),
             myactuator(a),
             currentCstr(cstr),
             currentCstrReq(o),
             currentSVVal(v),
             cmds(cl),
             waitAfter(msg(c,sv,startCstr(cstr,o,v''))),
             ctrlatts:AttributeSet >
msg(c,sv,newVal(v'))
=>
if satisfy(v',cstr)
then < c : ccid | mysv(sv),
                  myactuator(a),
                  currentCstr(cstr),
                  currentCstrReq(o),
                  currentSVVal(v'),
                  cmds(cl),
                  waitAfter(noMsg),
                  ctrlatts:AttributeSet >
     msg(sv,c,endCstr(cstr,o,noReason))
else (if cl == nil
      then < c : ccid | mysv(sv),
                        myactuator(a),
                        currentCstr(cstr),
                        currentCstrReq(o),
                        currentSVVal(v'),
                        cmds(cl),
                        waitAfter(noMsg),
                        ctrlatts:AttributeSet >
           msg(sv,c,endCstr(cstr,o,COANoSuccess))
      else < c : ccid | mysv(sv),
                        myactuator(a),
                        currentCstr(cstr),
                        currentCstrReq(o),
                        currentSVVal(v'),
                        cmds(rest(cl)),
                        waitAfter(msg(c,sv,startCstr(cstr,o,v))),
```

```
                          ctrlatts:AttributeSet >
             msg(a,c,issueCmd(first(cl)))
        fi)
  fi .
endm

mod CONTROLLER-TEST is
  inc CONTROLLER .

  op Ctrl : -> CtrlCid .

  op mycontroller : -> Object .
  eq mycontroller =
     < o("MyController") : Ctrl | mysv(o("MyStateVar")),
                                  myactuator(o("MyActuator")),
                                  waitAfter(noMsg) > .

endm
```

A.1.7 Estimator

```
*****************************************************************************
***                          Estimator                                    ***
***                                                                       ***
***                                                                       ***
*****************************************************************************

mod ESTIMATOR is
  inc ATTRIBUTES .
  inc ESTIMATOR-SENSOR-INTERFACE .
  inc STATE-VARIABLE-ESTIMATOR-INTERFACE .

  sort EstimatorCid .
  subsort EstimatorCid < Cid .

  *** Attributes
  op state : StateValue -> Attribute .

  *** converts a measurement into a state value
```

```
  op verifyState : Measurement  -> StateValue .

  vars sens est sv : Oid .
  var estcid  : EstimatorCid .
  var m : Measurement .
  var v : StateValue .

  rl[receiveMeasurementUpdStateReq]:
  < est : estcid | mysensor(sens),
                   mysv(sv),
                   waitAfter(noMsg),
                   state(v),
                   estatts:AttributeSet >
  msg(est, sens, newMeasurement(m))
  =>
  < est : estcid | mysensor(sens),
                   mysv(sv),
                   waitAfter(msg(est,sens,newMeasurement(m))),
                   state(verifyState(m)),
                   estatts:AttributeSet >
  msg(sv,est,updStateReq(verifyState(m))) .


  rl[receiveAckUpdateStateVar]:
  < est : estcid | mysv(sv),
                   waitAfter(msg(est,sens,newMeasurement(m))),
                   estatts:AttributeSet >
  msg(est,sv,ackUpdStateReq(v))
  =>
  < est : estcid | mysv(sv),
                   waitAfter(noMsg),
                   estatts:AttributeSet > .

endm


mod ESTIMATOR-TEST is
  inc ESTIMATOR .

  op Estimator : -> EstimatorCid .
```

```
  op myestimator : -> Object .

  eq myestimator =
     < o("MyEstimator") : Estimator | mysv(o("MyStateVar")),
                                      mysensor(o("MySensor")),
                                      state(uk),
                              waitAfter(noMsg) > .
endm
```

A.1.8 The Grid

```
*** Description of grid and grid ops

fmod GRID is
  pr NAT .

  sorts Loc LocSet .
  subsort Loc < LocSet .

  op mt : -> LocSet .
  op __ : LocSet LocSet -> LocSet [ assoc comm id: mt ] .

  op `(_`,_`) : Nat Nat -> Loc .

  op member : Loc LocSet -> Bool .
  eq member(l:Loc, l:Loc ls:LocSet) = true .
  eq member(l:Loc, ls:LocSet) = false [owise] .

  sort Dir .
  ops N W E S : -> Dir .

  op newLoc : Nat Nat Loc Dir LocSet -> Loc .
  op next : Nat Nat Loc Dir -> Loc .

  vars ht wd x y : Nat .

  eq newLoc(ht, wd, (x,y) , d:Dir, blocked:LocSet)
      = ( if member(next(ht,wd,(x,y),d:Dir),blocked:LocSet)
```

```
          then (x,y)
          else next(ht,wd,(x,y),d:Dir)
          fi )  .

  eq next(ht,wd, (x,y), N) =
       if (s y < ht) then (x, s y) else (x,y) fi .

  eq next(ht,wd, (x,y), S) =
       if (0 < y) then (x, sd(y,1)) else (x,y) fi .

  eq next(ht,wd, (x,y), E) =
       if (s x < wd) then (s x, y) else (x,y) fi .

  eq next(ht,wd, (x,y), W) =
       if (0 < x) then (sd(x,1), y) else (x,y) fi .

  op dir : Nat -> Dir .
  eq dir(0) = N .
  eq dir(90) = E .
  eq dir(180) = S .
  eq dir(270) = W .
endfm


fmod GRID-TEST is
  inc GRID .
  op B : -> LocSet .
  eq B = (1,1) (2,2) (3,3) .

endfm
```

A.2 Rover System

A.2.1 Position and Heading Interfaces

```
******************************************************************************
***               Special PosAndHead Interfaces                            ***
```

```
***                                                                        ***
******************************************************************************

******************************************************************************
***          PosHeadController-PosHeadActuator-Interface                   ***
***                                                                        ***
******************************************************************************

fmod PHCONTROLLER-PHACTUATOR-INTERFACE is
  inc CONTROLLER-ACTUATOR-INTERFACE .

  ops drive turn : -> Command .
endfm


******************************************************************************
***          Rover-PosHeadActuator-Interface                               ***
***                                                                        ***
******************************************************************************

fmod ROVER-PHACTUATOR-INTERFACE is
  inc DEVICE-ACTUATOR-INTERFACE .

  ops drive turn : -> MsgBody .
endfm

******************************************************************************
***           Rover-PosHeadSensor-PosHeadSensor-Interface                  ***
***                                                                        ***
******************************************************************************

fmod ROVER-PHSENSOR-INTERFACE is
  inc DEVICE-SENSOR-INTERFACE .
  inc GRID .

  op posAndHead : Loc Dir -> SensorValue .
endfm

******************************************************************************
***        PosHeadEstimator-PosHeadSensor-Interface                        ***
```

```
***                                                                    ***
**************************************************************************

fmod PHESTIMATOR-PHSENSOR-INTERFACE is
  inc ESTIMATOR-SENSOR-INTERFACE .
  inc GRID .

  sort PosAndHeadMeasurement .
  subsort PosAndHeadMeasurement < Measurement .

  *** For simplicity same as  PosAndHeadStateValue
  op _`,_dir_ : Nat Nat Dir  -> PosAndHeadMeasurement .
endfm

**************************************************************************
***   Special PosAndHead State Value                                   ***
***                                                                    ***
**************************************************************************

*** used in POSANDHEAD-ESTIMATOR
***         POSANDHEAD-STATE-VARIABLE
***         POSANDHEAD-CONTROLLER
***         POSANDHEAD-ESTIMATOR
fmod POSANDHEAD-STATE-VALUE is
  inc GRID .
  inc STATE-VALUE .

  sort PosAndHeadStateValue .
  subsort PosAndHeadStateValue < StateValue .

  op  _`,_dir_ : Nat Nat Dir  -> PosAndHeadStateValue .

  op _@x : PosAndHeadStateValue -> Nat .
  op _@y : PosAndHeadStateValue -> Nat .
  op _@d : PosAndHeadStateValue -> Dir .

  vars x y : Nat .
  var d : Dir .

  eq (x,y dir d)@x = x .
```

```
  eq (x,y dir d)@y = y .
  eq (x,y dir d)@d = d .
endfm
```

A.2.2 Position and Heading State Variable

```
****************************************************************************
***                  PosAndHeadStateVariable                             ***
***  has as value a triple (Nat, Nat, Dir) (Location/Direction)          ***
***  knows its estimator, controller                                     ***
****************************************************************************

mod POSANDHEAD-STATE-VARIABLE is
  inc POSANDHEAD-STATE-VALUE .
  inc STATE-VARIABLE .

  sort PosAndHeadSVCid .
  subsort PosAndHeadSVCid < SVCid .

endm

mod POSANDHEAD-STATE-VARIABLE-TEST is
  inc POSANDHEAD-STATE-VARIABLE .

  op PosAndHeadStateVar : -> PosAndHeadSVCid .

  op myphsv : -> Object .
  eq myphsv =
     < o("MyPosAndHeadStateVar") : PosAndHeadStateVar |
               myctrl(o("MyPosAndHeadCtrl")),
               myest(o("MyPosAndHeadEstimator")),
               req(o("NoOid")),
               waitAfter(noMsg),
               cstr(noCstr),
               val((0,0 dir(E))) > .
endm
```

A.2.3 Position and Heading Controller

```
***************************************************************************
***          Position and Heading Controller                             ***
***************************************************************************
fmod POSANDHEAD-CONSTRAINT is
  pr NAT .
  pr GRID .
  inc CONSTRAINT .

  sort PosAndHeadConstraint .
  subsort PosAndHeadConstraint < Constraint .

  op ___ : Nat Nat Dir -> PosAndHeadConstraint .
                  *** for now: constraint is new location and direction
                  *** subsequent versions will allow for delta-x,
                  *** delta-y, delta-theta, max velocity, etc.

  op _@x : PosAndHeadConstraint -> Nat .
  op _@y : PosAndHeadConstraint -> Nat .
  op _@d : PosAndHeadConstraint -> Dir .

  vars x y : Nat .
  var d : Dir .

  eq (x y d)@x = x .
  eq (x y d)@y = y .
  eq (x y d)@d = d .
endfm

mod POSANDHEAD-CONTROLLER is
  pr CONTROLLER .
  inc POSANDHEAD-CONSTRAINT .
  inc POSANDHEAD-STATE-VARIABLE .
  inc PHCONTROLLER-PHACTUATOR-INTERFACE .

  sort PosAndHeadCtrlCid .
  subsort PosAndHeadCtrlCid < CtrlCid .

  *** Specification of 'satisfy'
```

```
var phv :   PosAndHeadStateValue .
var phcstr : PosAndHeadConstraint .

eq satisfy(phv,phcstr) =
   if (phv)@x == (phcstr)@x and (phv)@y == (phcstr)@y
       and (phv)@d == (phcstr)@d
   then true
   else false
   fi .

*** Auxiliary functions to determine the COA
op turnFromTo : Dir Dir -> Command .
op turnFor : Nat -> Command .
op driveFor : Nat -> Command .
op cmdList :  Command -> CommandList .


*** Specification of 'coa'
*** Right now simple strategy has 3 steps
*** 1. try to achieve new x loc by:
***    a) possibly turning, then b) driving along x-axis
*** 2. try to achieve new y loc by
***    a) possibly turning, then b) driving along y-axis
*** 3. try to achieve new direction by turning

eq coa(phv,phcstr) =
   if (phv)@x < (phcstr)@x
   then (if (phv)@y == (phcstr)@y
         then (cmdList(turnFromTo((phv)@d, E)) ;
               cmdList(driveFor(sd((phcstr)@x,(phv)@x))) ;
               cmdList(turnFromTo(E, (phcstr)@d)))
         else (if (phv)@y < (phcstr)@y
               then (cmdList(turnFromTo((phv)@d, E)) ;
                     cmdList(driveFor(sd((phcstr)@x,(phv)@x))) ;
                     cmdList(turnFromTo(E, N)) ;
                     cmdList(driveFor(sd((phcstr)@y,(phv)@y))) ;
                     cmdList(turnFromTo(N, (phcstr)@d)) )
               else (cmdList(turnFromTo((phv)@d, E)) ;
                     cmdList(driveFor(sd((phcstr)@x,(phv)@x))) ;
                     cmdList(turnFromTo(E, S)) ;
```

```
                   cmdList(driveFor(sd((phv)@y,(phcstr)@y))) ;
           cmdList(turnFromTo(S, (phcstr)@d)))
            fi)
      fi)
     *** use 'sd' since _-_ not defined for Nat
     *** _-_ also ambiguous with __
else (if (phv)@y == (phcstr)@y
      then (cmdList(turnFromTo((phv)@d, W)) ;
            cmdList(driveFor(sd((phcstr)@x,(phv)@x))) ;
            cmdList(turnFromTo(W, (phcstr)@d)))
      else (if (phv)@y < (phcstr)@y
            then (cmdList(turnFromTo((phv)@d, W)) ;
                  cmdList(driveFor(sd((phcstr)@x,(phv)@x))) ;
                  cmdList(turnFromTo(W, N)) ;
                  cmdList(driveFor(sd((phcstr)@y,(phv)@y))) ;
                  cmdList(turnFromTo(N, (phcstr)@d)))
            else (cmdList(turnFromTo((phv)@d, W)) ;
                  cmdList(driveFor(sd((phcstr)@x,(phv)@x))) ;
                  cmdList(turnFromTo(W, S)) ;
                  cmdList(driveFor(sd((phv)@y,(phcstr)@y))) ;
          cmdList(turnFromTo(S, (phcstr)@d)))
            fi)
     fi)
fi .

*** driving for n steps yields in a command list with n "drive" commands
var n : Nat .
eq cmdList(driveFor(0)) = nil .
eq cmdList(driveFor(n)) = (drive ; cmdList(driveFor(sd(n,1)))) .

*** since the rover turns in 45 degrees, clockwise, turning from one
*** direction to another is a sequence of 45 degree turns in clockwise
*** direction
*** dir-diff computes the number of 45 degree turns necessary
var d d' : Dir .
op dir-diff : Dir Dir -> Nat .
eq dir-diff(E,E) = 0 .
eq dir-diff(E,S) = 2 .
eq dir-diff(E,W) = 4 .
eq dir-diff(E,N) = 6 .
```

```
    eq dir-diff(S,S) = 0 .
    eq dir-diff(S,W) = 2 .
    eq dir-diff(S,N) = 4 .
    eq dir-diff(S,E) = 6 .
    eq dir-diff(W,W) = 0 .
    eq dir-diff(W,N) = 2 .
    eq dir-diff(W,E) = 4 .
    eq dir-diff(W,S) = 6 .
    eq dir-diff(N,N) = 0 .
    eq dir-diff(N,E) = 2 .
    eq dir-diff(N,S) = 4 .
    eq dir-diff(N,W) = 6 .

    op turnFor : Nat -> Command .
    eq cmdList(turnFromTo(d,d'))
       = cmdList(turnFor(dir-diff(d,d'))) .
    eq cmdList(turnFor(0)) = nil .
    eq cmdList(turnFor(n)) = (turn ; cmdList(turnFor(sd(n,1)))) .
    eq cmdList(turn ; turn ; turn ; turn ; turn ; turn ; turn ; turn)
       = nil .
endm

mod POSANDHEAD-CONTROLLER-TEST is
  inc POSANDHEAD-CONTROLLER .

  op PosAndHeadCtrl : -> PosAndHeadCtrlCid .

  op noOid : -> Oid .
  op myphctrl : -> Object .
  eq myphctrl =
    < o("MyPosAndHeadCtrl") : PosAndHeadCtrl |
                mysv( o("MyPosAndHeadStateVar")) ,
                myactuator( o("MyPosAndHeadActuator")) ,
                currentCstr(noCstr) ,
                currentCstrReq(noOid) ,
                currentSVVal(uk) ,
                cmds(noCmd) ,
                waitAfter(noMsg) > .
endm
```

A.2.4 Position and Heading Actuator

```
*****************************************************************************
***          Position and Heading Actuator                                ***
*****************************************************************************
fmod POSANDHEAD-ACTUATOR is
  inc ACTUATOR .
  inc PHCONTROLLER-PHACTUATOR-INTERFACE .
  inc ROVER-PHACTUATOR-INTERFACE .

  sort PosAndHeadActuatorCid .
  subsort PosAndHeadActuatorCid < ActuatorCid .

  *** Specification of ‘executeCmd’
  eq executeCmd(drive) = drive .
  eq executeCmd(turn) = turn .

endfm

mod POSANDHEAD-ACTUATOR-TEST is
  inc POSANDHEAD-ACTUATOR .

  op PosAndHeadActuator : -> PosAndHeadActuatorCid .

  op myphactuator : -> Object .
  eq myphactuator =
     < o("MyPosAndHeadActuator") : PosAndHeadActuator |
                                          mydevice(o("MyRover")),
                                          myctrl(o("MyPosAndHeadCtrl")),
                                          executedCmd(noCmd) > .
endm
```

A.2.5 Position and Heading Sensor

```
*****************************************************************************
***          Position and Heading Sensor                                  ***
*****************************************************************************
fmod POSANDHEAD-SENSOR is
  inc SENSOR .
```

```
  inc DEVICE-SENSOR-INTERFACE .
  inc PHESTIMATOR-PHSENSOR-INTERFACE .
  inc ROVER-PHSENSOR-INTERFACE .

  sort PosAndHeadSensorCid .
  subsort PosAndHeadSensorCid < SensorCid .

  *** Specification of createMeasurement(sensorValue)
  var sval : SensorValue .
  vars x, y : Nat .

  eq createMeasurement(posAndHead((x,y),d:Dir)) =  ((x,y dir (d:Dir))) .
endfm

mod POSANDHEAD-SENSOR-TEST is
  inc POSANDHEAD-SENSOR .

  op PosAndHeadSensor : -> PosAndHeadSensorCid .

  op myphsensor : -> Object .
  eq myphsensor =
     < o("MyPosAndHeadSensor") : PosAndHeadSensor |
                                          mydevice(o("MyRover")),
                                          myest(o("MyPosAndHeadEstimator")),
                                          measurement(noMeas) > .
endm
```

A.2.6 Position and Heading Device / Rover

```
mod ROVER is
  inc DEVICE .
  inc PHCONTROLLER-PHACTUATOR-INTERFACE .
  inc ROVER-PHACTUATOR-INTERFACE .
  inc ROVER-PHSENSOR-INTERFACE .
  inc GRID .

  sort RoverCid .
  subsort RoverCid < DeviceCid .
```

```
*** Attributes
op grid : Nat Nat LocSet -> Attribute .
          *** Dimension of Grid:  ht  wd  blocked
op pos_  : Loc -> Attribute .
op hd : Nat -> Attribute .     *** degrees clockwise from grid north
sort Status .
ops driving turning idle : -> Status .
op st :  Status -> Attribute .


vars r a s : Oid .
vars x y x' y' ht wd h : Nat .
var blocked : LocSet .
var rcid : RoverCid .

rl[drive]:
< r : rcid | myactuator(a),
             mysensor(s),
             pos (x,y),
             hd(h),
             st(idle),
             grid(ht,wd,blocked),
             ratts:AttributeSet >
msg(r,a,drive)
=>
( if (h rem 90) == 0
  then < r : rcid | myactuator(a),
                    mysensor(s),
                    pos (x,y),
                    hd(h),
                    st(driving),
                    grid(ht,wd,blocked),
                    ratts:AttributeSet >
  else < r : rcid | myactuator(a),
                    mysensor(s),
                    pos (x,y),
                    hd(h),
                    st(idle),
                    grid(ht,wd,blocked),
                    ratts:AttributeSet >
```

```
       msg(s,r,sensorValues(posAndHead((x,y),dir(h))))
 fi ) .

crl[driving]:
< r : rcid | mysensor(s),
             pos (x,y),
             hd(h),
             st(driving),
             grid(ht,wd,blocked),
             ratts:AttributeSet >
=>
< r : rcid | mysensor(s),
             pos (x',y'),
             hd(h),
             st(idle),
             grid(ht,wd,blocked),
             ratts:AttributeSet >
msg(s,r,sensorValues(posAndHead((x',y'),dir(h))))
if  (x',y') := newLoc(ht,wd,(x,y), dir(h), blocked) .


rl[turn]:
< r : rcid | myactuator(a),
             pos (x,y),
             hd(h),
             st(idle),
             grid(ht,wd,blocked),
             ratts:AttributeSet >
msg(r,a,turn)
=>
< r : rcid | myactuator(a),
             pos (x,y),
             hd(h),
             st(turning),
             grid(ht,wd,blocked),
             ratts:AttributeSet >  .

rl[turning]:
< r : rcid | mysensor(s),
             pos (x,y),
```

```
              hd(h),
              st(turning),
              grid(ht,wd,blocked),
              ratts:AttributeSet >
  =>
  < r : rcid | mysensor(s),
              pos (x,y),
              hd((h + 45) rem 360),
              st(idle),
              grid(ht,wd,blocked),
              ratts:AttributeSet >
  msg(s,r,sensorValues(posAndHead((x,y),dir(((h + 45) rem 360))))) .

endm


mod ROVER-TEST is
  inc GRID-TEST .
  inc ROVER .

  op rov : -> Object .
  op Rover : -> RoverCid .

  eq rov = < o("MyRover") : Rover | myactuator(o("MyPosAndHeadActuator")),
                                    mysensor(o("MyPosAndHeadSensor")),
                                    pos(0,0),
                                    hd(90),
                                    st(idle),
                                    grid(5,5,B) > .
endm
```

A.2.7 Rover System

```
*****************************************************************************
***                    PosAndHeadSystem                                   ***
***  includes all necessary basic and application-specific modules        ***
*****************************************************************************

mod SYSTEM is
```

```
  inc GRID .
  inc POSANDHEAD-STATE-VARIABLE-TEST .
  inc POSANDHEAD-CONTROLLER-TEST .
  inc POSANDHEAD-ACTUATOR-TEST .
  inc POSANDHEAD-SENSOR-TEST .
  inc POSANDHEAD-ESTIMATOR-TEST .
  inc ROVER-TEST .
endm
```