Formal Checklists
for Remote Agent Dependability

Grit Denker[a,*], and Carolyn L. Talcott[a,**]

[a] *SRI, Menlo Park, California, USA*
{grit.denker,carolyn.talcott}@sri.com

Abstract

Remote agents used in Deep Space Missions such as rovers or solar airplanes must function autonomously over a prolonged time during planetary exploration. The Mission Data System (MDS) framework has been developed to address design and deployment of these complex systems. We are using the Maude environment to develop a formal framework with methods and supporting tools for increasing the dependability of MDS space systems. This is done by developing formal executable specifications of the MDS framework and its mission-specific adaptations and providing a set of formal checklists (formal analysis suites) that can be used to achieve better predictability and dependability. In this paper we present our formal model of the MDS framework, an adaptation for a remote rover and preliminary checklists for remote agents.

Keywords: Rewriting logic, goal-oriented, model-based, formal checklist

1   Introduction

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 [1] 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.   Goal-achiever software has the responsibility of elaborating high-level 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.

The main objective of our project is to develop a formal framework with methods (called *formal checklists*) and supporting tools for increasing the dependability of goal-oriented operation of space systems.   To clarify the basic ideas, we have modeled in Maude [2] a simplified version of the MDS Framework. Although simple, the modular structure of the Maude model follows that of the MDS components and filling in details to obtain a complete specification of the MDS architecture will be straightforward. A remote rover, called *SCRover*, is being developed at University of Southern California (USC) as a mission-specific adaptation of MDS. A formal executable specification of (a simplified version of the) SCRover adaptation has been developed in Maude through rover specific extensions of the abstract MDS components and the specification of a rover device model. A first set of checklist items has been defined and applied to the SCRover model using the execution, search and model-checking capabilities of Maude.

The remainder of this paper is organized as follows. Section 2 is a very brief review of relevant aspects of Maude syntax and tools. Section 3 discuss the notion of a goal and model based system design, gives an overview of the MDS framework and describes our specification of the MDS framework. Section 4 describes our specification of the SCRover adaptation. Section 5 presents initial checklist ideas and shows their application to the SCRover specification. Section 6 concludes and discusses future work. The Maude code and a more detailed technical report can be found at
http://www.csl.sri.com/users/denker/remoteAgents/.

2   About Maude

Maude [3,2] is a multi-paradigm executable specification language based on rewriting logic [4,5]. The Maude interpreter is very efficient, allowing prototyping of quite complex test cases. Maude also provides efficient built-in search and model checking capabilities. Maude is reflective [6,7] providing a meta-level module that reflect both the syntax and semantics of Maude. Using reflection the user can program special purpose execution and search strategies, module transformations, analyses, and user interfaces. 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`.

We briefly summarize the syntax of Maude that is used in our case study. 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`
- *system* modules, that are rewrite theories specifying concurrent systems; they are declared with the syntax `mod...endm`

These modules have an *initial model semantics*. 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 followed by the operator name, the argument and result sorts. An operator may have mixfix syntax, with the name containing '`_`'s marking the argument positions. Equational axioms are introduced with the keyword `eq` (or `ceq` for conditional equations) followed by the two terms being declared equal separated by the equality sign `=`. Rewrite rules are introduced with the keyword `rl` (or `crl` for conditional rules) followed by an optional rule label, and terms corresponding to the premises and conclusion of the rule separated by the rewrite sign `=>`. Variables appearing in axioms, rules (and commands) may be declared globally using keyword `var` or `vars`, or "inline" using the variable name and its sort separated by a colon, for example `n:Nat` is a variable named `n` of sort `Nat`. Rewrite rules are not allowed in functional modules.

We model the various components of the MDS framework and its adaptations using the Maude notation and conventions for concurrent objects. A (snapshot of a) system state (sort `Configuration`) is a multiset of objects (sort `Object`) and messages (sort `Msg`). The multiset union operator for configurations is denoted with empty syntax (juxtaposition) and (by definition of multiset) is associative and commutative.

An object has the the form `< O : C | att-1, ... , att-n >` where `O` is an object identifier (sort `Oid`), `C` is a class identifier (sort `Cid`), and `att-1`, ..., `att-n`

are attributes (sort `Attribute`).

The above is axiomatized in the module `CONFIGURATION` which is part of the standard Maude library. A typical system configuration will consist of several objects and messages. The dynamic behavior of a concurrent object system is then axiomatized by specifying rewrite rules for each class that determine, for example, what an object does in response to a message.

For the purpose of the MDS application, we defined a module called `MYCONF` specifying a constructor `op o : String  -> Oid` that converts a string into an object identifier, a sort `MsgBody` for message contents, and message constructors `op msg : Oid Oid MsgBody -> Msg` and `op noMsg : -> Msg`. This gives a standard form to messages for convenient axiomatization of a general interaction framework.

3   The MDS Architecture
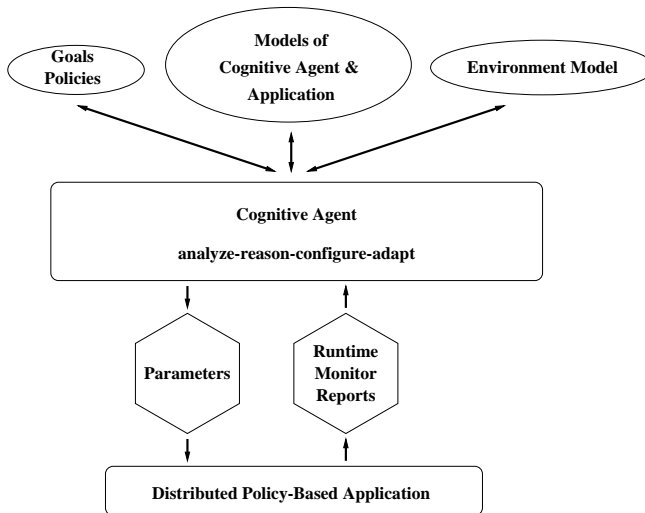
3.1   Model and Goal-Based System Design

Fig. 1. Adaptive System Design

The MDS framework is an example of a general approach, we call *model-and goal-based system design* (see Figure 1), to the design of distributed, situation-aware systems that can adapt to dynamically changing requirements and environments. A key element of this approach is the use of formal models and formal representation of the relations of models with each other and with system parameters and observables. These models fall into three main areas:

(i) Formal models of the underlying system infrastructure, including models of relevant aspects of devices and of software components of the envisioned system.

(ii) High-level goals and policies that express various requirements of the envisioned systems, including end-to-end functionality, performance, security, service classes, and quality of service, as well as administrative, computational, or physical distribution requirements.

(iii) Environment models describing threat and failure models, expected usage patterns, traffic load, or physical environment constraints among others.

A congnitive agent provides goal-elaboration and situation analysis services that use the models along with sensors (passive monitors or active probes) and policy-based configuration services, to achieve the overall goals of situation-aware and adaptive systems. A goal elaboration service analyzes the current situation using its models of system and environment state and computes initial values for or constraints, i.e., low-level policy specifications, on system parameters that are expected to meet the goals and enforce the policies. A policy-based configuration service uses these parameter restrictions to compute actual system configurations. Sensors observe system execution events and inform the cognitive agent about ongoing system behavior. Reports are analyzed to determine whether the observed behavior meets expectations according to the current system and environment models. If there are problems, adaptation of the models is attempted, using justifications for current expectations produced by the goal-elaboration service. When models are updated, or new goals are specified, the goal-elaboration service recomputes system parameters and policies on the basis of changed models and goals. In addition to the MDS framework, we are currently investigating applicability of this approach to cognitive networking, and real-time, embedded systems.

The remainder of this section is organized as follows. First we give a brief overview of the MDS architecture components and connectoins. Then we describe the Maude modules that formalize this architecture—the interfaces and messages exchanged between components, and classes that model the common structure of each component. The work to date has focused on specification of models and simple goals. Specifying goal elaboration is the next step. In MDS, a scheduler plays the role of policy-based configuration service.

3.2 MDS Overview

The Mission Data System (MDS) [1] and its precursor remote agent architectures [8] 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 and (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. Mission goals describe the desired outcome of a mission operation so that the overall mission will be successfully completed. Mathematically, a goal is a prioritized constraint on the value of a state variable during a time interval.
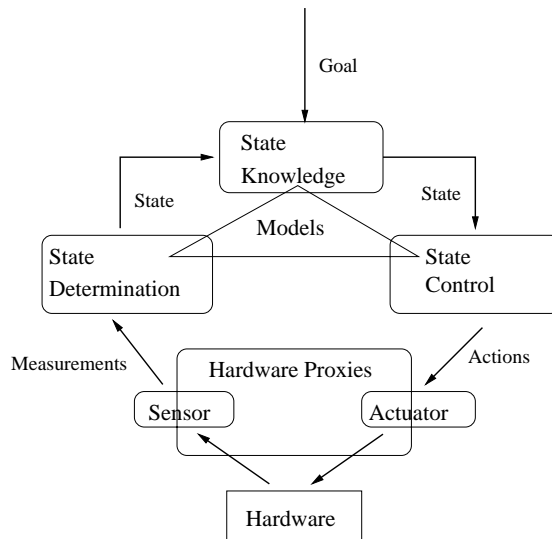

Fig. 2. The MDS state-based architecture

The main ingredients of the MDS architecture are depicted in Figure 2 (taken from [1], 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 control the system must be made explicit as values of state variables. State variables are the only way to access a system state. Thus, a key part of system design is the choice of state variables. 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. 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). Estimators interpret measurements

and update values of state variables. 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 has not yet been satisfied, or to tackle the next goal.

3.3 The MDS Architecture in Maude

Our formal model is based on publications describing the MDS architecture, including [1], documents available from the SCRover testbed artifact repository (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 Jet Propulsion Laboratory (JPL) and the SCRover team at USC.

We model the MDS architecture with the following six modules, each defining a class: `STATE-VARIABLE`, `CONTROLLER`, `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). We will discuss the `STATE-VARIABLE` module in some detail, to give an idea of the specification, and briefly summarize the key features of the remaining modules.

We model communication between MDS components using asynchronous message passing. Some of the message exchanges are meant to model method invocations (the object-oriented analog of function call). This is specified using a special attribute `waitAfter` that takes a message as parameter and conditional rules that use the value of the `waitAfter` attribute to control what messages can be received. Each method invocation message has a corresponding set of possible replies. Separate modules are used to specify the communication interfaces—the messages exchanged—between components. Figure 3 shows the Maude classes and the messages that are sent between components with arrows indicating the direction of message flow.

The rules of these classes specify information and control flow roughly as follows. A goal request is made to the state variable, which forwards it to the controller. The controller determines the course of action and issues commands for the device via the actuator. The resulting device state is read by the sensor and sent to the estimator. The estimator creates measurements out of the sensor values and sends update requests to the state variable. When the state variable receives the 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 entity
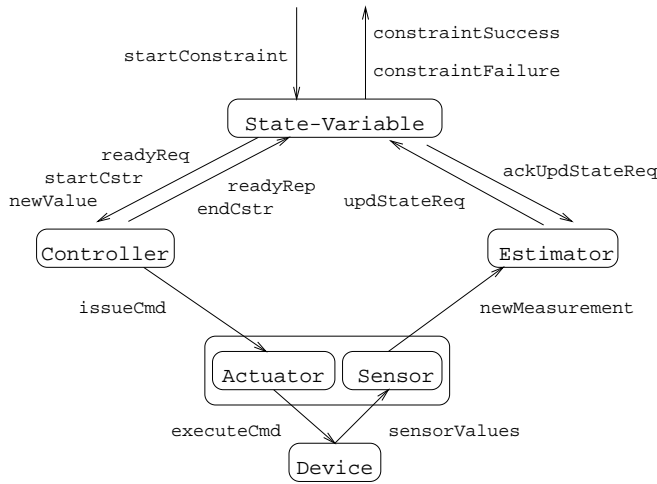
Fig. 3. MDS Component Message Exchange

specified in the goal request.

3.4 Maude Specification of Interfaces

We restrict our discussion to state variable interfaces with the environment and controller. Other interfaces have a similar structure.

State Values.

State values are computed by estimators, communicated in messages, tested for constraint satisfaction by controllers, and used by goal-elaborators. The module STATE-VALUE declares a sort for state values, a subsort for *unknown* state values, and a constant uk to represent the fact that the value of a state variable may be unknown. Here and elsewhere the sort and class hierarchy of the Maude model mirrors the class hierarchy of the MDS framework, to facilitate relating the model to the implementation.

```
fmod STATE-VALUE is
  sort StateValue .
  sort UnknownStateValue .
  subsort UnknownStateValue < StateValue .

  op uk : -> UnknownStateValue .
endfm
```

State Variable/Environment Interface.

The environment sends start constraint requests to 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. The module CONSTRAINT declares a sort Constraint, for constraints, together with a constant noCstr of sort Constraint that stands for the constraint that is always satisfied, analogous to the boolean value true. The module REASON declares a sort Reason, for reasons. The following module specifies the messages for the interface between state variables and the environment as follows.

```
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 .
endfm
```

State Variable/Controller Interface.

Before requesting that the controller starts a constraint, the state variable must ask if the controller is ready for a new constraint. If it receives a positive reply, it will forward the constraint, including the requester identity (from the environment) and its current state value. The following module specifies the messages for the interface between state variables and its controller.

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

  *** to controller
  op readyReq : -> MsgBody .        *** to controller
  op startCstr : Constraint Oid StateValue -> MsgBody .
  op newVal : StateValue -> MsgBody .

  *** to state variable
  op readyRep : Bool -> MsgBody .
  op endCstr : Constraint Oid Bool Reason -> MsgBody .
endfm
```

3.5 Maude Specification of MDS Components

3.5.1 Maude Specification of State Variable

The STATE-VARIABLE module includes modules for attributes and interfaces with the controller and estimator. It declares the state variable class identifier sort SVCid to be a subsort of the predefined Cid defined in CONFIGURATION. This way the state variable class is a subclass of the generic object class. Each specific state variable class will have its own class identifier sort that is in turn a subsort of

SVCid. This way rules defined for state variable objects in general will apply to state variable objects of specific subclasses. Additional attributes declared for state variables include a current value, the constraint that is being processed, if any, and an object identifier to remember the requester of that constraint. Finally a reason constant, notReady, is declared to use as a constraint failure reason in case the controller replies negatively to a ready request.

```
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 determines the behavior of state variables. We show only the two rules used for handling a start constraint request.

```
vars sv o o' ctrl : Oid .
var svcid : SVCid .
var cstr cstr' : Constraint .
var svatts : AttributeSet .
var v : StateValue .
```

A startConstraint message can only be accepted if the state variable is not waiting for some reply, that is, the value of waitAfter is noMsg. In this case it records the requests and sends its controller a readyReq message.

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

A state variable can only accept a readyReply from its controller if it is waiting after a startConstraint. If the reply is positive the constraint is forwarded to the controller and the state variable maintains its waiting status. Otherwise it reports failure to the requester and becomes ready to accept another constraint.

```
rl[forwardConstraint]:
```

```
< sv : svcid | svatts, myctrl(ctrl), req(o), cstr(cstr), val(v),
              waitAfter(msg(sv,o,startConstraint(cstr))) >
msg(sv,ctrl,readyRep(b))
=>
if b
then < sv : svcid | svatts, myctrl(ctrl), req(o), cstr(cstr),
                    val(v), waitAfter(msg(sv,ctrl,readyRep(b)))) >
     msg(ctrl,sv,startCstr(cstr,o,v))
else < sv : svcid | svatts, myctrl(ctrl), req(o), cstr(cstr),
                    val(v), waitAfter(noMsg) >
     msg(o,sv,constraintFailure(cstr,notReady))
fi .
```

Note that the state variable rules use a class identifier variable `svcid` of sort `svCid` rather than a specific constant of this sort. This is so that the rule will apply to specific state variables with class identifier whose sort is a subsort of `svCid`. This is a standard technique for modeling subclassing in Maude.

The remaining state variable rules concern state value updates and constraint end reports. The state variable can receive a state value update request from its estimator at any time. When an update arrives the state variable stores the new value, acknowledges the receipt of the update request to the estimator and informs the controller about the new value.

A state variable can only accept a message `endConstraint(cstr,r)` from its controller if it is waiting for a report for `cstr`. If the reason `r` is `noReason`, then `ConstraintSuccess(cstr)` is sent to the constraint requester, otherwise, `ConstraintFailure(cstr,r)` is sent.

3.5.2   Maude Specification of Other Components

In the following we give an overview of the functionality of other MDS components. Complete Maude specifications of all MDS components can be found at http://www.csl.sri.com/users/denker/remoteAgents/.

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.

When a controller receives a ready 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 (indicated `waitAfter(noMsg)`). Otherwise it will respond negatively.

When a controller receives a message `startConstraint(cstr,o,v)` from a state variable (having previously reported that it is ready) it saves the constraint, the requester, and the current state value. The controller decides whether

the current state value satisfies the constraint using an operation `satisfy`. If the current state value already satisfies the constraint, it replies to the state variable with a successful `endCstr` message. If the current state does not satisfy the constraint, the controller determines the course of action by calling its `coa` function, issues the first command and stores the rest of the command list.

If the controller is in the process of satisfying a constraint and it receives a message with a new value, `newVal(v)`, it first determines if the constraint is satisfied by the new value. If so, an `endCstr` message is sent to the state variable indicating success. If the constraint is not satisfied and the controller has no more commands to issue, an `endCstr` message is sent to the state-variable with the reason `COANoSuccess` indicating that the course of action determined by the controller was not successful. In both of the above cases the controller sets its `waitAfter` attribute to `noMsg` to allow new constraints to be processed. Otherwise the controller issues the next command in its stored command list and remains in the processing constraint state.

Both the `satisfy` and `coa` operations of the controller must be defined separately for each particular adaptation of the MDS architecture.

Actuator.

When an actuator, `act`, receives a message `msg(act,ctrl,issueCmd(cmd))`, it sends `msg(d,act,executeCmd(cmd))`, to its associated device `d`. The operation `executeCmd` converts the controllers command into a form acceptable to the device.

Sensor.

When a sensor, `sens`, receives `msg(sens,d,sensorValues(sval))`, from its device `d`, it sends

$$msg(est,sens,newMeasurement(createMeasurement(sval)))$$

to its associated estimator, `est`. The operation `createMeasurement` is used to convert sensed values into measurements.

Estimator.

When an estimator `est` receives `msg(est,sens,newMeasurement(m))` from the sensor, it sends `msg(sv,est,updStateReq(verifyState(m)))` to the associated state variable `sv`. The function `verifyState` is used to estimate the state value. The estimator waits for an acknowledgment from the state variable before accepting further measurements by storing the new measurement message in its `waitAfter` attribute. This ensures that values are received by the state variable

in the same order that they were sent by the estimator. It also means that the state variable is obliged to accept and acknowledge all updates.

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 explicit 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 MDS adaptation. Therefore, the generic device module only provides sort declarations for the general device class.

4   The SCRover Executable Specification

Our example of a small remote agent system is a rover moving on a grid where some of the grid positions contain obstacles. The rover can rotate clockwise in increments of 45 degrees, and it can move in the direction it is heading if this direction is a multiple of 90 degrees, and if the adjacent position in that direction is not blocked or off the grid.

The module `GRID` specifies the grid, declaring sorts `Loc` (pairs of natural numbers `(x,y)`), `LocSet` (sets of locations), and `Dir` (symbolic names for the compass point directions, `N,S,E,W`). Functions are defined to convert symbolic directions to degrees (`dir(N) = 0`), and to compute the new position after a move, `newLoc`.

The module `ROVER` specifies the grid based rover. The rover knows about the grid it operates on (its `grid` attribute specifies the dimension of the grid in terms of height, width and the set of blocked locations), its own position (`loc` containing its grid coordinates) 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
  ...
  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 .
  ...
endm
```

The rover can receive a drive 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. It will also be unable to move if there is an obstacle occupying the target grid position. In this case, the rover will simply report to the sensor its current position and heading. The rover can also receive a turn command from its actuator. In this case it executes the command and then reports its current position and heading to its sensor.

The Position and Heading State Value and Interfaces.

We use one state variable to model the rover state, called a position and heading state variable. As the name indicates, the value domain for this state variable consists of triples giving the x and y grid coordinates, and the direction in which the rover is headed. The module POSANDHEAD-STATE-VALUE specifies this value domain. In particular, it declares a subsort PosAndHeadStateValue of StateValue and a constructor

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

In the rover adaptation, the controller-actuator, device-actuator, device-sensor, and estimator-sensor interfaces must be further refined to specify the data to be communicated between components. Two commands drive and turn are added to the controller-actuator interface. These are also added to the actuator-device interface as message bodies, using overloading. A constructor

```
          posAndHead : Loc Dir -> SensorValue
```

is added to the device-sensor interface for the rover to report values to the sensor.

The Position and Heading State Variable.

As discussed above, an important part of designing an MDS remote agent system is identifying the state variables and determining 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.

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

  sort PosAndHeadSVCid .
  subsort PosAndHeadSVCid < SVCid .
```

```
endm
```

This module defines a subclass of `STATE-VARIABLE` by declaring a subsort `PosAndHeadSVCid` of `SVCid` for the `POSANDHEAD-STATE-VARIABLE` class identifier. Similar conventions are used to define the position and heading subclasses for the remaining components (controller, actuator, sensor, and estimator).

The Position and Heading Controller.

This module extends the generic controller by defining the syntax of position and heading constraints, giving equations for constraint satisfaction, and giving equations for the course-of-action function `coa`. Position and heading constraints are elements the sort `PosAndHeadConstraint`, a subsort of `Constraint`. A position and heading constraint is simply a triple consisting of the juxtaposition of two natural numbers (the intended location to be understood as (x, y)) and a direction. A position-and-heading value satisfies a position-and-heading constraint if their corresponding coordinates and direction are the same.

The algorithm for determining the course of action to achieve a given constraint implements 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.

(i) 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.

(ii) Second the final y-position of the goal is achieved. The controller determines whether the rover needs to proceed along the y-axis in south or north direction, issues the required turning commands and then determines the number of driving steps for the rover.

(iii) Finally, the controller determines how many more turns need to be issued to get the rover headed in the desired direction.

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 does not end up in the desired position, because a drive command resulted in no change of position because of a block.

The Position and Heading Actuator, Sensor, and Estimator.

The specifications for the position and heading actuator, sensor and estimator classes are very simple. Thus, the actuator takes a command from the controller and transforms it into the correct format for execution in the device. Because we have chosen the same names for commands and actuator-rover messages this is just a sort conversion. The operations converting sensor values into measurements (`executeCmd`) and measurements into values (`verifyState`) are also defined to be sort conversions, simply returning their arguments viewed as elements of a different sort.

5   Towards Formal Checklists

The checklist idea refines the *Maude Formal Methodology* [9] by providing steps to follow for a particular family of specifications, in this case specifications of autonomous space systems based on the MDS framework. The essence of this methodology is that a little formality can a long way, and different levels of assurance can be obtained by different levels of effort: developing a formal model; execution of test scenarios; light-weight analysis such as search and model-checking; and theorem proving.

We illustrate the MDS checklist ideas in the context of our SCRover specification.

- L0. Check that the specification is well-formed.
  This check is implemented simply by loading the specification into Maude and ensuring there are no problems reported.

- L1. Execution level checks that representative configurations exhibit expected and desired behavior. For each test case
  · Define a test module and define an initial system configuration.
  · Extend the test module with definitions of goals to be checked. This should cover all the primitive goals.
  · For each goal specify expected outcomes as well as situations that should not arise using predicates defined on configurations.
  · Execute the test scenarios (initial configuration plus goal) using one of the default rewriting strategies, and check the final state against the specified expectations.
  · Use search to determine if all executions meet expectations.

- L2. Analysis level checks
  · Extend the test module with definitions of system invariants, i.e., properties that should hold of all system configurations.
  · Using search (built-in or special purpose), check invariants for the test

scenarios.

· Extend the test module with definitions of temporal properties (expressed in Maude's LTL) that the system should satisfy.

· Check the temporal properties using model-checking.

Note that each checklist level builds upon the previous in the sense that the modules used at one level are extended to obtain the modules used at the next level. Furthermore, the modules produced to define the scenarios and properties serve as documentation of the expectations and what was checked.

Also, when developing an MDS application the checklist procedure should be applied to each component individually as well as to the composed system. In the case of components, "goal" is replaced by messages in the components interface. Sequences of such messages will be sent by a generic tester component that plays the role of the tested components environment.

As systems get more complex, with multiple devices, and multiple concurrent goals, the checklist process will be correspondingly more complex, but it will build on the basic elements. With more experience we expect that steps such as 'define goals', and 'define invariants', can be partially automated, using general architectural principles, diagrams describing expected usage, and formalizations of device specifications.

We have applied the checklists to the SCRover specification. Each rover system component has a test module defining the components initial state, and test executions sending the component interface messages. For example, the state variable test module is the following.

```
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
```

The full system test module imports the component test modules and defines the initial configuration to consist of the six components in their initial states. A constructor for constraint request messages is also defined.

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

```
  op sys : -> Configuration .
  eq sys =  myphsv myphctrl myphactuator myphsensor myphest rov  .
  op mkm : Nat Nat Dir -> Msg .
  vars x y : Nat . var d : Dir .
  eq mkm(x,y,d) =  msg(o("MyPosAndHeadStateVar"), o("MyRequester"),
                       startConstraint((x y d))) .
  op ic : Nat Nat Dir -> Configuration .
  eq ic(x,y,d) = sys mkm(x,y,d) .
endm
```

We specified the scenarios and expected outcomes (expressed as a subconfiguration of the final state) shown in Figure 4.

```
scenario        expected outcome(s)
------------    -------------------
1. ic(1,0,E)    < o("MyRover") : Rover | atts, pos(1,0),hd(90)  >
                msg(o("MyRequester"), o("MyPosAndHeadStateVar"),
                    constraintSuccess(1 0 E))

2. ic(2,0,E)    < o("MyRover") : Rover | atts, pos(2,0),hd(90)  >
                msg(o("MyRequester"), o("MyPosAndHeadStateVar"),
                    constraintSuccess(2 0 E))

3. ic(1,2,E)    msg(o("MyRequester"), o("MyPosAndHeadStateVar"),
                    constraintFailure(1 2 E, COANoSuccess))))

4. ic(1,0,E)    2 outcomes: for x in {1,2}
   mkm(2,0,E)   < o("MyRover") : Rover | atts, pos(x,0),hd(90)  >
                msg(o("MyRequester"), o("MyPosAndHeadStateVar"),
                    constraintSuccess(1 0 E))
                msg(o("MyRequester"), o("MyPosAndHeadStateVar"),
                    constraintSuccess(2 0 E))
```

Fig. 4. Rover scenarios

For scenarios 1-3, execution gives the one expected outcome, and for scenario 4 execution gives one of the expected outcomes. Searching for all terminal states using the command

```
search sys mkm(1,0,E) mkm(2,0,E) =>! C:Configuration
```

yields the two expected outcomes. Such checks can be automated fairly easily using reflection.

In an earlier version of the model, the state variable accepted constraints while it was waiting for a report from the controller for a previous constraint. In this case, we expected two kinds of outcome for scenario 4.

(i) The rover 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

(ii) The rover 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 using the search graph, we found two
problems: the state variable forgot the constraint that was being processed,
and the controller didn't remember that it had replied yes to a ready request,
and hence could agree to starting several constraints. These were problems
in our model, but similar problems could easily arise in implementations.
They illustrate the importance of analyzing the model for validation of the
model itself. It also emphasizes the importance of making formal connections
between the model and the informal designs and implementation.

To partially automate checking outcomes we defined parameterized success
and failure patterns and used the model-checker to look for the expected
pattern. This could also be done using search. The advantage to model-
checking in the current setting is that when the question is formulated so that
a positive result from the model-checker is a counter-example, a rule trace can
be extracted as a simple trace of the execution.

6   Conclusions and future work

This paper reports on initial progress in our project to use Maude as the basis
for developing *formal checklists* for Deep Space Mission software built using
the MDS framework.

We have presented a simplified version of the MDS architecture to test
our understanding of the structure and constraints. In order to make further
elaboration and adaptation easier, some effort was made to take a system-
atic specification approach to the control structures for controller, actuator,
sensor, estimator, and state variable interactions with each other and with
the framework scheduler. We also studied how mission specific devices, their
models, controllers and estimators can be modeled, and specified a simple ver-
sion of the SCRover. Simple execution unearthed several small problems in
the control flow. In multiple goal scenarios, search of the state space yielded
unexpected outcomes. Examination of the search graph provided the needed
information to determine the problems. In the process of developing these first
models, we have also discovered gaps in the informal documentation (filled by
discussion with the experts).

Future work includes refining our MDS framework model to include more
details about the components such as timelines for state variables and mea-
surement histories for sensors. In addition we will extend the SCRover specifi-

cation with the missing state variables (power, camera, range finder) and refine the models to adequately reflect the hardware specifications and behavior characterics. Also, formal connections between the model and implementation (for example header files and scheduling specifications) will be developed to relate model analysis results to code. The next big step is to model goals, goal nets and goal elaboration. This will involve modeling time points—temporal variables that express sequentialization and ordering between goals as well as durations. A Maude based language for specification of goals and goal elaboration strategies will be developed. Then checklist items will be developed and automated to check the feasibility of goals and the consistency of goal nets with respect to the physical device models.

References

[1] D. Dvorak, R. Rasmussen, G. Reeves, and A. Sacks. Software Architecture Themes In JPL's Mission Data System. In *IEEE Aerospace Conference, USA*, 2000.

[2] http://maude.cs.uiuc.edu. The Maude Homepage.

[3] M. Clavel, S. Eker, P. Lincoln, and J. Meseguer. Principles of Maude. In *Rewriting Logic Workshop'96*, number 4 in Electronic Notes in Theoretical Computer Science. Elsevier, 1996. http://www.elsevier.nl/locate/entcs/volume4.html.

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

[5] José Meseguer. Rewriting logic and Maude: A wide-spectrum semantic framework for object-based distributed systems. In S. Smith and C.L. Talcott, editors, *Formal Methods for Open Object-based Distributed Systems, FMOODS 2000*, pages 89–117. Kluwer, 2000.

[6] Manuel Clavel. *Reflection in General Logics, Rewriting Logic, and Maude*. PhD thesis, University of Navarre, 1998.

[7] M. Clavel and J. Meseguer. Reflection and strategies in rewriting logic. In *Rewriting Logic Workshop'96*, number 4 in Electronic Notes in Theoretical Computer Science. Elsevier, 1996. http://www.elsevier.nl/locate/entcs/volume4.html.

[8] 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, 1998.

[9] G. Denker, J. Meseguer, and C. L. Talcott. Formal specification and analysis of active networks and communication protocols: The Maude experience. In *DARPA Information Survivability Conference and Exposition (DISCEX 2000)*, volume 1, pages 251–265. IEEE, 2000.