Validating Behavioral Component Interfaces in Rewriting Logic

Einar Broch Johnsen, Olaf Owe, and Arild B. Torjusen [1]

Department of Informatics, University of Oslo, Norway

Abstract

Many distributed applications can be understood in terms of components interacting in an open environment such as the Internet. Open environments are subject to change in unpredictable ways, as other applications may arrive, evolve, or disappear. In order to validate components in such environments, it can be useful to build a simulation environment which reflects this highly unpredictable behavior. In this paper, the validation of components with respect to behavioral interfaces is considered. Behavioral interfaces specify semantic requirements on the observable behavior of components, expressed in an assume-guarantee style. In our approach, a rewriting logic model is transparently extended with the history of all observable communication, and metalevel strategies are used to guide the simulation of environment behavior. Over-specification of the environment is avoided by allowing arbitrary environment behavior within the bounds of the assumption on observable behavior, while the component is validated with respect to the guarantee of the behavioral interface.

Key words: Validation, components, behavioral interfaces, simulation strategies, rewriting logic, meta-programming

1 Introduction

This paper suggests an application of rewriting logic [17] to test the behavior of software units in *open distributed environments* such as the Internet. An open environment is an environment in which various other software units exist, and little or no information about these units is available. A distributed environment is an environment in which communication is asynchronous. Reasoning in this setting is intrinsically difficult, partly due to the non-determinism caused by distribution, but more characteristically due to the unknown and evolving open environment.

It is a major challenge to predict the behavior of components evolving in open distributed environments, in order to ensure and maintain behavioral properties such as safety, availability, quality of service, and various forms of fault tolerance. Formal approaches to system verification, such as Hoare logic, type checking, and

[1] Email: einarj@ifi.uio.no, olaf@ifi.uio.no, aribraat@ifi.uio.no

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

model checking, depend on knowing the implementation details of the system components, including those in the open environment. Approaches based on testing simulate an environment in which the system can be subjected to test runs. In contrast to verification methods, testing cannot generally ensure that components are always well-behaved, but testing may still give revealing insights into a component's behavior. However, the problem of conformance testing for software units in open distributed environments is not resolved [25]. This paper aims to show how open environments can be mimicked by underspecified formal descriptions based on *observable behavior* in order to validate the behavior of software units in open distributed environments. Model-based testing in the early development stages makes the testing process more effective [19].

Object orientation is the leading framework for concurrent and distributed systems, recommended by the RM-ODP [12] and used in, e.g., .Net and Corba. In this paper, we model distributed components by objects which asynchronously exchange messages. The models are executable in the rewriting logic system Maude [4], which has facilities for simulation, model checking, and verification. To allow black-box validation, we use requirement specifications in terms of observable behavior. Observable behavior is specified using *behavioral interfaces* [13,14] which describe component services available to the environment.

The contribution of this paper is to define an executable framework for validating the observable behavior of models in the open distributed setting. For this purpose, behavioral interfaces are captured in rewriting logic and combined with a standard rewriting logic model of asynchronously communicating objects. Furthermore, the executable platform in Maude is extended with validation facilities in a transparent way. Rewriting logic is *reflective* [3,5] in a mathematically precise manner: it is possible to reason formally about reflective rewriting inside rewriting logic itself, and to execute reflective specifications at the Maude *metalevel*. The use of reflection is essential to our approach, allowing for guided search and system monitoring in a modular, composable, and hierarchical way. Reflection may be used to define execution strategies for an executable object model, for example a *nondeterministic* execution strategy is proposed in [15]. Reflective specifications support a layered architecture where several specifications may be given at each level. Reflection can be used to extend a system model with, e.g., logging facilities [24]. In this paper, we transparently extend an executable specification with its history of observable communications at the metalevel, and define execution strategies at the metalevel which are influenced by requirements on the communication history. One strategy is used to mimic open environments and another to test the executable model. The two strategies are combined in order to enable assume-guarantee testing.

Paper overview: Sect. 2 presents a formalism for behavioral interfaces. Sect. 3 summarizes rewriting logic and the Maude tool. Sect. 4 develops metalevel strategies for monitoring and testing executable Maude models. A strategy for simulation of open environments is presented in Sect. 5 and it is shown how this can be utilized in a test scenario. Sect. 6 discusses related and future work.

2 Behavioral Interfaces

An open distributed system (ODS) can be represented by components or objects that run in parallel and communicate asynchronously by means of remote method calls. The implementation details of the components may be unknown, so reasoning must rely on abstract specifications of the system's components. We assume that components come equipped with *behavioral interfaces* that instruct us on how to use them. Furthermore, as a component may be used for multiple purposes, it can come equipped with *multiple* interfaces. This section presents a formalism for viewpoints by a notion of generic interfaces with behavioral requirements, restricted to safety aspects. For further details about this work, see [13,14].

Black-box specifications of concurrent components may be expressed in terms of *observable behavior*, i.e., the time sequence of input and output to the program. This fits well with the notion of encapsulation; only visible operations are considered at the specification level. An execution can be represented by a sequence of communication events, which is infinite in the case of non-terminating executions. However, infinite sequences are not easy to reason about. To avoid infinite sequences, specifications may be expressed in terms of the finite initial segments of the executions, capturing the abstract states of components during execution. These sequences are commonly referred to as histories [6] or traces [11]. Prefix-closed sets of executions express safety properties in the sense of Alpern and Schneider [1].

Finite sequences. We consider an abstract data type $\mathsf{Seq}[T]$ of finite sequences parameterized by a type T. Functions over sequences will be defined by means of convergent sets of equations, using the empty sequence, ε, and right append, $_;_ : \mathsf{Seq}[T] \times T \to \mathsf{Seq}[T]$, as sequence constructors. We let "_" denote argument positions of functions with mix-fix notation.

We define projection, $_/_ : \mathsf{Seq}[T] \times \mathsf{Set}[T] \to \mathsf{Seq}[T]$, and an "ends with" relation, $_\mathbf{ew}_ : \mathsf{Seq}[T] \times \mathsf{Set}[T] \to \mathsf{Bool}$, using one equation for each constructor case:

$$\varepsilon/S = \varepsilon \qquad\qquad \varepsilon\ \mathbf{ew}\ S = \mathsf{false}$$
$$(t;x)/S = \mathbf{if}\ x \in S\ \mathbf{then}\ (t/S);x\ \mathbf{else}\ t/S \qquad\qquad (t;x)\ \mathbf{ew}\ S = x \in S$$

The notation $\#t$ denotes the length of t.

2.1 Semantics

Let Obj and Mtd be unbounded sets of object identifiers and method names. Let Data be a set of data values, including Obj. In this paper, we conventionally let $o_1, o_2 \in \mathsf{Obj}$ and $m \in \mathsf{Mtd}$. A *communication event* has the form

$$msg\ \mathbf{from}\ o_1\ \mathbf{to}\ o_2$$

where *msg* consists of Data. This term is considered an *output event* of o_1 and an *input event* of o_2. For observable events, o_1 and o_2 are distinct. The sets of observ-

able input and output events of an object o are denoted IN_o and OUT_o, respectively, and are by definition disjoint. Their union is denoted $INOUT_o$.

An *alphabet* for an object o is a subset of $INOUT_o$. An alphabet of o may cover certain aspects of the communication of o; and in the next section we introduce syntax for statically defined alphabets. A *trace set* $\mathcal{T}_\alpha \subseteq \mathsf{Seq}[\alpha]$ is a prefix-closed set of well-formed sequences.

Definition 2.1 [Specification] A specification Γ is a triple $\langle o, \alpha, \mathcal{T} \rangle$ where (1) $o \in \mathsf{Obj}$ is an object identifier, (2) α is a possibly infinite alphabet for o, and (3) $\mathcal{T}$ is a trace set over α.

For any specification Γ, we can derive a *communication environment* $\mathcal{E}(\Gamma)$ of objects communicating with the object of Γ. In an ODS setting, we generally think of the communication environment as unbounded. Since the specification Γ does not need to cover all aspects of the behavior of o, we say that Γ is an *interface specification* (of o).

In the following we consider object-oriented distributed systems where communication is achieved through remote methods calls. In order to achieve asynchronous communication, we model a method call through two events: the event representing the initiation of a call, and the event representing its completion. For a call from o_1 to o_2 of method m, the initiation event is generated by the caller o_1 and is represented by $invoc(m)$ **from** o_1 **to** o_2, and the completion event is generated by the callee o_2 and represented by $comp(m)$ **from** o_2 **to** o_1. To simplify the exposition, we abstract from parameter values in this paper. In order to increase readability, we represent these events by $o_1 \rightarrow o_2.m$ and $o_1 \leftarrow o_2.m$, respectively.

As we consider asynchronously communicating objects, a caller may communicate while (passively) waiting for a completion and a callee may communicate while performing a method, and other events can be observed in between the initiation and completion of any given call. When we consider the history of observable behavior, every completion event must be preceded by a corresponding invocation, which gives rise to the following notion of well-formedness:

$$\begin{aligned}
wf(\varepsilon) &= true \\
wf(t;(o \rightarrow o'.m)) &= wf(t) \\
wf(t;(o \leftarrow o'.m)) &= wf(t) \wedge \#(t/o \rightarrow o'.m) \geq \#(t/o \leftarrow o'.m)
\end{aligned}$$

where $\#(t/o \rightarrow o'.m)$ is the length of the trace t restricted to invocation events of the method m from o to o', and similarly for *comp* events.

Definition 2.2 [Refinement] A specification $\langle o, \alpha, \mathcal{T} \rangle$ of o is said to *refine* another specification $\langle o, \alpha', \mathcal{T}' \rangle$ of o if $\alpha' \subseteq \alpha$ and $\forall t \in \mathcal{T} \,.\, t/\alpha' \in \mathcal{T}'$.

Thus, refinement corresponds to subset on projected trace set in the sense that $\{t/\alpha' \mid t \in \mathcal{T}\} \subseteq \mathcal{T}'$. Composition of specifications may be introduced to define partial components or system aspects in the sense of distributed services [13,14].

2.2 Syntax

Interface specifications may be given in a generic manner. Generic specifications are referred to as *behavioral interfaces*. An object may support a number of interfaces. As Maude does not provide a syntax for specification of observable behavior, statically defined alphabets, nor methods (not even with Full Maude), we introduce a syntax for observable behavior by means of object-oriented interfaces:

interface F (⟨context parameters⟩)
  inherits $F_1, F_2, \ldots, F_m$
begin
with *cointerface*
    op $m_1(\ldots)$
      ...
    op $m_n(\ldots)$
  spec
  where <auxiliary function definitions>
end

Interfaces can have context parameters, which typically describe the minimal environment, representing static links needed by objects that support the interface. An initiation and a completion event is associated with each method declaration (ranging over method parameters). In the specification formula, we use the keyword "*this*" to denote the object supporting the interface.

Mutual dependency. We let objects be typed by interfaces. By identifying a type for the caller, the *cointerface*, we restrict the objects that may call the methods of this interface, while allowing *this* object to call cointerface methods. This opens up for interaction with a caller during execution of a method. In an implementation language, access to the *caller* may be provided by means of an explicit parameter as in Maude, or an implicit parameter as in Creol [15]. This gives strong typing in an asynchronous setting. Semantically a cointerface declaration augments the alphabet of this interface, as the methods available through the cointerface supported by the caller are added.

Inheritance. Multiple inheritance is allowed for interfaces, but cyclic inheritance graphs are not allowed. In a subinterface, we can declare additional methods and behavioral constraints. A cointerface declaration applies to the locally declared methods. If an interface F is declared with an inheritance clause, the alphabets of the super-interfaces are included in the alphabet of F. Specifications are inherited by conjunction, when restricted to the relevant alphabets of the super-interfaces. An interface will always refine its super-interfaces.

Definition 2.3 [Interface alphabet] The alphabet of an object o with respect to the interface F, denoted $\alpha_{o:F}$, is defined as the set of events of the form

$invoc(m)$ **from** o' **to** o and $comp(m)$ **from** o **to** o' for m declared in F,
$invoc(m)$ **from** o **to** o' and $comp(m)$ **from** o' **to** o for m declared in (or inherited by) the cointerface,

and any event in $\alpha_{o:F'}$ where F' is a super-interface of F.

Definition 2.4 [Interface specification] Let h range over histories. For each super-interface F' with specification **spec** $P'(h)$, the conjunct $P'(h/\alpha_{this:F'})$ is added to the specification of this object.

Assume-guarantee predicates. In ODS, the environment in which an object exists is subject to change, and specifications are relative to an assumed behavior of the environment. We adapt the assume-guarantee specification style [16] to the setting of observable behavior. Assumptions should set restriction on the inputs and guarantees on the outputs. However, it is often difficult to formulate assumptions and guarantees separately, since requirements to outputs may depend on earlier input, and requirements to inputs may depend on earlier output. Instead we use a single predicate P which relates input and output events, and extract an assumption part and a guarantee part from P:

Definition 2.5 [Assume-guarantee] Let IN and OUT denote the sets of input and output events for *this* interface. The trace set given by the specification **spec** $P(h)$ is $\{h \mid G(h)\}$, where the assumption part A and the guarantee part G are defined by:

$$\begin{aligned} A(\varepsilon) &= \mathsf{true} \\ A(h;x) &= A(h) \wedge (x \in IN \wedge P(h) \Rightarrow P(h;x)) \\ G(\varepsilon) &= \mathsf{true} \\ G(h;x) &= G(h) \wedge (A(h;x) \Rightarrow P(h;x)) \end{aligned}$$

Note that both sets $\{h \mid G(h)\}$ and $\{h \mid A(h)\}$ are prefix-closed, and that their intersection is the largest (prefix-closed) trace set contained in $\{h \mid P(h)\}$.

Assumptions are the responsibility of the objects in the environment. The assumption part ensures that each input is OK, assuming no earlier violation. Guarantees are the responsibility of the object supporting the interface; they are guaranteed when the assumption holds. The guarantee part ensures that each output is OK, assuming the assumption holds. Thus, an actual environment is required to refine the trace set given by A, and an implementation of the interface is required to refine the trace set given by G.

2.3 Example: A Minimal Interface

Behavioral interfaces are illustrated through the example of the dining philosophers. A table object informs a philosopher of the identity of the philosopher's left neighbor and provides units of food. A philosopher may borrow and return its neighbor's chopstick. Interaction between the philosophers and the table is restricted by interfaces. This results in a clear distinction between internal methods and methods externally available to other objects typed by the *cointerface*. Here, each philosopher owns one chopstick and must borrow another from a neighbor before eating. Hence, philosophers have both active and passive behavior. Strong typing and cointerfaces guarantee that only philosophers may call the methods *borrowStick* and *returnStick*.

```
interface Phil                  interface Table
begin                           begin
with Phil                       with Phil
   op borrowStick                  op seat(out neighbor:Phil)
   op returnStick                  op eat
   ⟨specification⟩              end
end
```

Denote by *caller* an arbitrary *Phil* object in the environment of *this Phil* object, as required by the cointerface. The alphabet of *Phil* is given by the events:

$$caller \rightarrow this.borrowStick \qquad caller \leftarrow this.borrowStick$$
$$this \rightarrow caller.borrowStick \qquad this \leftarrow caller.borrowStick$$

and similar events for *returnStick*. We define the following specification in *Phil*:

$$\begin{aligned}
&\textbf{spec} \quad 0 \leq lent(h) \leq 1 \wedge 0 \leq borrowed(h) + requested(h) \leq 1\\
&\textbf{where}\ lent(h) = \#(h/ \leftarrow this.borrowStick) - \#(h/ \rightarrow this.returnStick)\\
&\quad borrowed(h) = \#(h/this \leftarrow borrowStick) - \#(h/this \rightarrow returnStick)\\
&\quad requested(h) = \#(h/this \rightarrow borrowStick) - \#(h/this \leftarrow borrowStick)
\end{aligned}$$

Here, *lent* captures the number of sticks lent to neighbors, *borrowed* the number of sticks the object has borrowed from its neighbors, and *requested* captures the number of unfulfilled borrow requests. The specification implies that a single boolean variable suffices to keep track of sticks given away. Thus, the assumption part of the specification reduces to

$$A_{Phil}(h;x) = A_{Phil}(h) \wedge (x \in \{\rightarrow this.returnStick\} \ \Rightarrow lent(h) > 0)$$

stating that the environment may not return more sticks than it has borrowed.

The two interfaces above are connected by introducing an interface *EatingPhil*, inheriting *Phil* and with a *Table* as a parameter, thereby providing initial environmental knowledge. The specification of *Phil* is strengthened by requiring that a philosopher must have two sticks to eat:

```
interface EatingPhil(table : Table) inherit Phil
begin
  spec   eating(h) ⇒ lent(h) = 0 ∧ borrowed(h) = 1
  where eating(h) = #(h/this ← eat) > #(h/this → eat)
end
```

Here, *eating* is true when *this* object is in the middle of eating. This interface does not strengthen the assumption inherited from *Phil*, i.e., $A_{EatingPhil}(h) = A_{Phil}(h) = \forall h' \leq h \cdot lent(h') \geq 0$.

3 Rewriting Logic and Maude

This section gives a brief introduction to rewriting logic [17] and Maude [4]. A rewrite theory is a 4-tuple $\mathcal{R} = (\Sigma, E, L, R)$, where the signature Σ defines the function

symbols of the language, E defines equations between terms, L is a set of labels, and R is a set of labeled rewrite rules. From a computational viewpoint, a rewrite rule $t \longrightarrow t'$ may be interpreted as a *local transition rule* allowing an instance of the pattern t to evolve into the corresponding instance of the pattern t'. Rewrite rules apply to fragments of a state configuration. If rewrite rules may be applied to non-overlapping fragments of the configuration, the transitions may be performed in parallel. Consequently, rewriting logic (RL) is a logic which easily captures concurrent change. A number of concurrency models have been successfully represented in RL [4,17], including Petri nets, CCS, Actors, Unity.

Informally, a state configuration in RL is a multiset of terms of given types. These types are specified in (membership) equational logic (Σ, E), the functional sublanguage of RL which supports algebraic specification in the OBJ [10] style. When modeling computational systems, configurations may include different system components modeled by terms of the different types defined in the equational logic. An RL object is a term of the type $\langle O : C \mid a_1 : v_1, \ldots, a_n : v_n \rangle$, where O is the object's identifier, C is its class, the a_i's are the names of the object's attributes, and the v_i's are the corresponding values [4].

RL extends algebraic specification techniques with rewrite rules to capture the dynamic behavior of a system, supplementing the equations defining the term language. Assuming that all terms can be reduced to normal form, rewrite rules transform terms modulo the defining equations of E. Each rule describes how a part of a configuration can evolve in one transition step:

$$\textbf{rl}\ [\mathit{label}] : \ \mathit{subconfiguration} \longrightarrow \mathit{subconfiguration}$$

Rewrite rules may be conditional. The condition is formulated as a conjunction of rewrites and equations which must hold for the main rule to apply:

$$\textbf{crl}\ [\mathit{label}] : \ \mathit{subconfiguration} \longrightarrow \mathit{subconfiguration}\ \textbf{\textit{if}}\ \mathit{condition}$$

An unconditional rule with an *if-then-else* expression as the right hand side may alternatively be given as two complementary conditional rules. Rules in RL may be formulated at a high level of abstraction, closely resembling a compositional operational semantics[18].

3.1 The Maude Metalevel

Rewriting logic is reflective in the sense that there is a finitely presented rewrite theory $\mathcal{U}$ that is *universal*, meaning that we can represent any finitely presented rewrite theory $\mathcal{R}$ (including $\mathcal{U}$ itself) in $\mathcal{U}$. Let C and C' be configurations and $\mathcal{R}$ be a set of rewrite rules. We write $\mathcal{R} \vdash C \rightarrow C'$ to express that C may be rewritten to C' in the rewrite theory $\mathcal{R}$. Informally, a configuration C and the set $\mathcal{R}$ of rewrite rules of a Maude specification may be represented as terms $\overline{C}$ and $\overline{\mathcal{R}}$ at the metalevel. Using this notation, we have the equivalence [3]:

$$\mathcal{R} \vdash C \rightarrow C' \Leftrightarrow \mathcal{U} \vdash \langle \overline{\mathcal{R}}, \overline{C} \rangle \rightarrow \langle \overline{\mathcal{R}}, \overline{C'} \rangle,$$

rl *[req-stick] :* $\langle$ *X:Ob* | *hungry : true, myS : yes, nbrS : no, nbr : Y* $\rangle$ $\longrightarrow$
$\langle$ *X :Ob* | *hungry: true, myS: yes, nbrS: req, nbr: Y* $\rangle$ *(invoc('borrowStick) from X to Y) .*

rl *[borrow] :* $\langle$ *X :Ob* | *hungry: false, myS: yes, nbrS: s, nbr: Y* $\rangle$
(invoc('borrowStick) from Z to X) $\longrightarrow$
$\langle$ *X :Ob* | *hungry: false, myS: no, nbrS: s, nbr: Y* $\rangle$ *(comp('borrowStick) from X to Z) .*

rl *[rcv-stick] :* $\langle$ *X :Ob* | *hungry: true, myS: yes, nbrS: req , nbr: Y* $\rangle$
(comp('borrowStick) from Y to X) $\longrightarrow$
$\langle$ *X :Ob* | *hungry: true, myS: yes, nbrS: yes, nbr: Y* $\rangle$ *.*

rl *[eat-req] :* $\langle$ *X :Ob* | *hungry: true, myS: yes, nbrS: yes, nbr: Y* $\rangle$ $\longrightarrow$
$\langle$ *X :Ob* | *hungry: true, myS: yes, nbrS: yes, nbr: Y* $\rangle$ *(invoc('eat) from X to 'table) .*

rl *[eat] :* $\langle$ *X :Ob* | *hungry: true, myS: yes, nbrS: yes, nbr: Y* $\rangle$
(comp('eat) from 'table to X) $\longrightarrow$
$\langle$ *X :Ob* | *hungry: false, myS: yes, nbrS: no, nbr: Y* $\rangle$ *(invoc('returnStick) from X to Y) .*

Figure 1. Rewrite rules capturing philosopher behavior.

which states that if a term C can be rewritten to a term C' in a rewrite theory $\mathcal{R}$, then the meta-representation of C in $\mathcal{R}$, $\langle\overline{\mathcal{R}},\overline{C}\rangle$, can be rewritten to the meta-representation of C' in $\mathcal{R}$, $\langle\overline{\mathcal{R}},\overline{C'}\rangle$, in the universal rewrite theory $\mathcal{U}$, and vice versa. Maude includes facilities to meta-represent a rewrite theory $\mathcal{R}$ and to apply rules from $\mathcal{R}$ to the meta-representation of a term C by so-called *descent functions*.

Metalevel rewrite rules may be used to select which rule from $\mathcal{R}$ to apply to which subterm of C. This is done by defining an interpreter function which takes as arguments a finitely presented rewrite theory $\mathcal{R}$, a term C, and a deterministic strategy S. Metalevel rewrite rules may further be used to modify a configuration or the rule set of a rewrite theory. Hence, metalevel rewriting can be used as a wrapper around a rewrite theory $\mathcal{R}$ in order to abstractly mimic a more elaborate rewrite theory $\mathcal{R}'$ extending $\mathcal{R}$. Further details on the theory and the use of reflection in RL and Maude may be found in [3,4,5].

3.2 Example: Implementation of the Philosophers

We introduce a Maude specification which implements the *EatingPhil* specification given in Sect. 2.3. Let O be a variable ranging over Oids, a philosopher object is defined as a RL object $\langle O : Ob | hungry : _, myS : _, nbrS : _, nbr : _\rangle$. The Boolean attribute *hungry* indicates whether the philosopher is hungry, the attributes *myS* and *nbrS* indicate the status of its chopsticks (*yes*,*no*,*req*), used to impose synchronization constraints on the specification, and *nbr* identifies the neighbor.

The philosopher interacts asynchronously with the environment by message passing. Internal actions are represented by a philosopher (asynchronously) passing messages to himself. A selection of rules from the specification is given in Fig. 1.

```
crl [exec-monitor] :
  ⟨M : MetaRep | curTerm : T, curModule : MOD, labels : L LS, failedRules : FR⟩
  ⟨History : H⟩                  ⟶
  if RES :: Result4Tuple then
    ⟨M : MetaRep | curTerm : getTerm(RES),curModule : MOD, labels : LS L,
                   failedRules : nil⟩
    ⟨History : H ; getNewMessages(T, getTerm(RES), MOD, H)⟩
  else
    ⟨M : MetaRep | curTerm : T, curModule : MOD, labels : LS L, failedRules : FR L⟩
    ⟨History : H⟩      fi
if  RES := metaXapply([MOD], T, L, none, 0, unbounded, 0)   ∧   #FR ≤ #LS.
```

Figure 2. The metalevel rewrite strategy $\mathcal{S}_{monitor}$ recording the communication history.

4 Monitoring and Testing Executable Models

The observable behavior of an executable model can be monitored by recording the *communication history* from an execution of the model: This can be done *transparently* with the aid of the Maude metalevel without modifying the original specification. We can further test that the execution conforms to the behavioral specification of the model by defining predicates that operate on the recorded history and block execution if a violation occurs.

To execute a specification at the metalevel, we develop a custom *strategy*; i.e., rewrite rules which apply to the meta-representation of the model. Thus the current state may be inspected in-between rewrites. This enables us to record a communication history while executing a specification: We can check whether the application of a rewrite rule results in the emission of a new message by comparing the metalevel representations of the configuration before and after the rule application.

The object $\langle M : \mathit{MetaRep} \,|\, \mathit{curTerm} : _, \mathit{curModule} : _, \mathit{labels} : _, \mathit{failedRules} : _\rangle$ is used to store the information needed to control consecutive metalevel rewrites. *curTerm* contains the meta-representation of the current configuration, *curModule* is the meta-representation of the name of the object-level module in which the rewrites will be performed, *labels* is a list of rule labels from this module, and *failedRules* contains a list of labels for rules that are not applicable to *curTerm*.

The object $\langle \mathit{History} : _\rangle$ has one attribute h which contains the actual communication history recorded at runtime as a message list. This object is distinct from the objects of the object-level model and is consequently not modified by any rewrite rule of the object level, nor is it needed to apply object-level rewrite rules.

The custom strategy $\mathcal{S}_{monitor}$ is implemented by a conditional rewrite rule $\mathit{exec} : \mathit{MetaRep} \times \mathit{History} \rightarrow \mathit{MetaRep} \times \mathit{History}$ (see Fig. 2). The actual term rewriting is performed by the built-in Maude function *metaXapply*, which returns a tuple from which we extract the rewritten term itself with *getTerm*. Note that whitespace in Maude denotes list concatenation: If L is a label and LS is a list of labels, then $L\ LS$ is a non-empty list of labels. The strategy applies rules from the *labels* list

to the metalevel configuration in *curTerm* in a round-robin fashion. (A strategy for random rule selection based on a pseudo-random number generator is given in [15].) If no rule is applicable, the execution will terminate. The auxiliary function *getNewMessages* compares the term T to the new system configuration, i.e., the result of applying the rule labeled L to T. If there are new communication messages in the new configuration, the attribute h of the history object is extended with the new messages. If there are several new messages, these are caused by concurrent actions and may therefore be added to the history in an arbitrary order.

We create the strategy $\mathcal{S}_{test}$ by extending $\mathcal{S}_{monitor}$ with functionality for checking whether a given rule application will lead to an illegal state, as specified by a predicate parameter. We consider predicates on communication histories as defined by behavioral interfaces. To obtain a compositional system, the predicate on the global history will be formulated as the conjunction of the requirement specifications of a number of behavioral interfaces, associated with different objects. Behavioral specifications for specific objects are represented by predicates on the global history, restricted to an appropriate subset of possible communication events.

The $\mathcal{S}_{test}$ strategy blocks further execution once the system attempts to reach an illegal state which would violate the predicate on the global history. If we want to test a particular object o against a behavioral specification $\langle o, \alpha, \mathcal{T}_\alpha \rangle$, the testing predicate can be expressed as $P(h) = h/\alpha \in \mathcal{T}_\alpha$. For behavioral requirements given as a predicate $P : \mathsf{Seq}[\alpha] \rightarrow \mathsf{Bool}$ defined by a convergent set of equations, membership in the trace set is effectively computable by reducing $P(h/\alpha)$ for the current global history h.

The $\mathcal{S}_{test}$ strategy is implemented in Maude by extending the conditional *exec* rule with a branch which checks the given predicate between each rewrite step and blocks execution if the predicate is invalidated. A Maude function *CheckPredicate* : *Pred* $\times$ *MsgList* $\rightarrow$ *Bool* is used for this purpose. A predicate is specified using a constant H which acts as a placeholder for the actual communication history. At run-time *CheckPredicate* parses the predicate specification against the actual history, calls any auxiliary predicates, and returns a boolean value indicating whether the history after the rewrite step would be in compliance with the predicate or not. If the execution is blocked by the strategy, the recorded history provides an error trace for the system run, describing how the specification was violated.

Example. Acceptable behavior for a system of philosophers behaving according to the behavioral interface defined in Sect. 2.3, can be expressed by the Maude operator *AccBeh*:

eq *AccBeh(nil)* = true
eq *AccBeh(H ; MSG from X to Y) = P(H/X ; MSG from X to Y)*

where P is the specification predicate of the *EatingPhil* interface, and where the notation h/X abbreviates $h/INOUT_X$. Since the predicate in the Maude specification is a *global* predicate that spans all objects, there is no need to pass the object identifier as a separate parameter to *AccBeh*. In addition, since *AccBeh* is checked for each input and output event incrementally, we do not need to use the guarantee and assumption parts defined in Sect. 2.2.

5 Simulation of Open Environments

Interface assumptions on the observable behavior may be used to generate arbitrary environment behavior within the limits imposed by the assumption predicate.

5.1 Syntactic Simulation of Open Environments

At the object-level, a rewrite theory is used to syntactically simulate the unknown environment. In an open environment, objects may be created and destroyed dynamically during execution. To mimic the open environment, we define a term containing a set *absIDs* of (abstract) object identifiers representing objects which may currently interact with the system: $\langle E : Envir \,|\, absIDs : _, sysIDs : _, seed : _\rangle$. The set *absIDs* will be used to generate input messages to the objects of the system. System objects are represented as a set *sysIDs* of pairs $Obj \times Set[Mtds]$ which consist of object identifiers and sets of method names corresponding to alphabets of interfaces. The *seed* attribute is used for message generation.

An open environment can be *simulated* such that the behavior of abstract objects is exclusively defined by the behavioral interfaces. The messages emitted by abstract objects are input to the real objects of the system.

In order to produce syntactically correct but arbitrary input to the system from objects in the environment, we need to select an object o from *sysIDs* and produce a message to o (either calling a method available in the interface of o or replying to a call from o found in the history). For this purpose, we use a pseudo-random number generator [15] and let the function $next : Nat \rightarrow Nat$ produce new seed values for the environment. Let the function $genMsg : Obj \times Obj \times Set[Msg] \times Nat \rightarrow Msg$ generate a new message *msg* to an object o with alphabet α in the system from an object in the environment, such that $msg \in \alpha$. The rewrite rule for message generation is given by:

$$\textbf{rl}\ [msg\text{-}gen] : \langle E : Envir \,|\, absIDs : o_1\, A, sysIDs : (o_2, \alpha)\, C, seed : X\rangle \longrightarrow \langle E : Envir \,|\, absIDs : o_1\, A, sysIDs : (o_2, \alpha)\, C, seed : next(X)\rangle\ genMsg(o_1, o_2, \alpha, X)$$

5.2 Semantic Simulation of Open Environments

At the metalevel, a rewrite theory is used to semantically simulate the unknown environment. Minimal behavioral requirements for open environments are given by assumptions in the system interfaces. Define a metalevel strategy $\mathcal{S}_{restrict}$ which *restricts* a rewrite system to behave according to a predicate on observable behavior. This strategy is similar to $\mathcal{S}_{test}$, but where $\mathcal{S}_{test}$ halts the execution when the application of an enabled rule would violate the predicate, $\mathcal{S}_{restrict}$ tries another enabled rule from the *labels* list of the *MetaRep* object instead. Open environments do not terminate; if no rewrite rule is applicable to any position of *curTerm*, the strategy changes the seed value and retries the rules.

The abstract environment specification can now be used as a *testbed* for an actual programmed component (see Fig. 3). Let $\mathcal{R}_1$ be an object-level set of rewrite

	Rule set:	Configuration:
Metalevel rewrite system:	$\mathcal{S}_{restrict}(P_1(h/\alpha_1))$ $\wedge\ \mathcal{S}_{test}(P_2(h/\alpha_2))$	$\overline{\mathcal{R}_1} \cup \overline{\mathcal{R}_2}, (\overline{\mathcal{C}_1}\ \overline{\mathcal{C}_2})$, $\langle History : h \rangle$
	$\downarrow$ Control	$\uparrow$ History logger
Object level rewrite system:	$\mathcal{R}_1 \cup \mathcal{R}_2$	$\mathcal{C}_1\ \mathcal{C}_2$

Figure 3. Reflective testing of observable behavior in open environments.

rules generating (and possibly garbage collecting) messages. Rules from $\mathcal{R}_1$ may be applied to a configuration $\mathcal{C}_1$ consisting of an *Envir* object. Let $\mathcal{R}_2$ be the object-level set of rewrite rules applicable to the concrete objects in a configuration $\mathcal{C}_2$, e.g., the given component, with synchronization constraints on the internal state. Let α_1 and α_2 be alphabets associated with the objects of $\mathcal{C}_1$ and $\mathcal{C}_2$, respectively, such that $\alpha_1 \subseteq \alpha_2$. Let P_1 and P_2 be predicates observationally specifying the environment and actual component, respectively. If several interfaces are considered, we let P_1 be the conjunction of the assumptions and P_2 the conjunction of guarantees, restricted to the relevant alphabets. The metalevel strategy $\mathcal{S}_{restrict}$ restricts rule application from $\mathcal{R}_1$ to acceptable environment behavior, providing an abstract, open environment which may behave in any way that does not violate the predicate P_1. We here combine two metalevel strategies which react differently to the violation of predicates: *restrict* will restrict rule application so that the communication history conforms to the predicate, and *test* will halt the execution and produce an error object if the predicate does not hold. By specifying one predicate that spans only messages from the objects of the component, and one that spans all objects, checking the former in *test* mode, and the latter in *restrict* mode, we can test whether our programmed component executes correctly provided that the environment does so.

5.3 *Execution of the Philosopher Example*

This test scenario was implemented in Maude by defining a metalevel rewrite rule *exec-test* similar to the one given in Fig. 2, but which combines the $\mathcal{S}_{restrict}$ and $\mathcal{S}_{test}$ strategies as described above. The metalevel specification was used to test the implementation of philosophers described in Sect. 3.2. The test configuration consisted of one concrete philosopher object, rules for a table object, and an environment of 4 additional philosophers, simulated as described in Sect. 5.1. The rewrite rules for philosopher behavior (Fig. 1) were checked against the *Phil* interface specification (Sect. 2.3) in $\mathcal{S}_{test}$ mode to test their correctness, whereas application of the *msg-gen* rule was restricted by the $\mathcal{S}_{restrict}$ strategy to conform to the assumption A_{Phil}.

When the number of applications of the *exec-test* rule of this non-terminating

specification was limited to 5000, the result (after 53494167 rewrites) was a trace of 355 messages to and from the object to be tested. We observe that if we deliberately introduce rules which violate the guarantee specification, the violation will be detected by the implementation. Furthermore, we notice that if we break the environment assumptions (e.g., by replacing the assumption predicate with the vacuous assumption *true*), this will cause a violation of the guarantee specification that will also be detected.

6 Related and Future Work

We do not attempt to fully survey the extensive literature on monitoring and testing here. Many previous history-based [8,19,22] and automata-based [2,21,23] approaches require specific and deterministic test cases to be defined. In contrast, we use *random testing* and assume-guarantee specifications to capture open environments, where environment behavior is arbitrary within the bounds of an assumption predicate. Invariant-driven strategies for Maude similar to our $\mathcal{S}_{restrict}$ have recently been proposed in [9], but that paper considers predicates on states rather than observable behavior and does not consider the application to open environments nor to testing. For open environments random testing within the bounds of minimal assumptions seems more attractive than deterministic tests.

The specifications of observable behavior considered in this paper are fairly easy to implement in rewriting logic. The specification language considered may be replaced by a more expressive language. For example, it would be interesting to combine our approach to open environment modeling with linear time temporal logic specifications on finite traces. An efficient algorithm in rewriting logic for the verification of such formulas has been given in [20].

7 Conclusion

The main contribution of this paper is to sketch an approach to the validation of black-box components in open environments by extending Maude models with a notion of observable behavior and related execution strategies. The paper shows how abstract specifications of open environments may be captured very naturally in a rewriting logic model extended with behavioral interfaces. The behavioral interfaces express safety requirements on the observable behavior of components. The approach is presented within a method-based, object-oriented setting, but may easily be adjusted to general asynchronous message passing. Due to the reflective character of rewriting logic, supported by Maude, it is possible to define execution strategies at the metalevel. In this paper, we have used this facility in four ways. First, a strategy is defined to non-deterministically generate arbitrary input to a system. Second, a strategy is defined to transparently introduce monitoring of a set of communication events. Third, a strategy is defined to restrict system input by semantic requirements on the observable behavior. Combining these strategies, the arbitrary behavior of open environments may be simulated within the bounds

of minimal assumptions. The separation of object-level and metalevel constraints facilitates experimenting with different assumptions on the environment. The same approach may also be used to execute a prototype model defined by its observable behavior, without deciding on its implementation details. Fourth, a strategy is defined to test whether an executable model is well behaved with respect to semantic requirements on the observable behavior. Combining all four strategies, we obtain abstract validation environments for models of components or distributed applications, in which the environment is unspecified but subjected to minimal observational requirements.

Acknowledgments. Eyvind W. Axelsen has contributed to the implementation of these ideas. The anonymous referees have provided useful comments.

References

[1] Alpern, B. and F. B. Schneider, *Defining liveness*, Information Processing Letters **21** (1985), pp. 181–185.

[2] Barbey, S., D. Buchs and C. Péraire, *A theory of specification-based testing for object-oriented software*, in: *Proc. Eur. Dependable Computing Conf. (EDCC2)*, LNCS **1150** (1996), pp. 303–320.

[3] Clavel, M., "Reflection in Rewriting Logic: Metalogical Foundations and Metaprogramming Applications," CSLI Publications, Stanford, California, 2000.

[4] Clavel, M., F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer and J. F. Quesada, *Maude: Specification and programming in rewriting logic*, Theoretical Computer Science **285** (2002), pp. 187–243.

[5] Clavel, M. and J. Meseguer, *Reflection in conditional rewriting logic*, Theoretical Computer Science **285** (2002), pp. 245–288.

[6] Dahl, O.-J., *Can program proving be made practical?*, in: M. Amirchahy and D. Néel, editors, *Les Fondements de la Programmation*, Institut de Recherche d'Informatique et d'Automatique, Toulouse, France, 1977 pp. 57–114.

[7] Dahl, O.-J., "Verifiable Programming," Prentice Hall, New York, N.Y., 1992.

[8] Doong, R.-K. and P. G. Frankl, *The ASTOOT approach to testing object-oriented programs*, ACM Trans. Softw. Eng. Methodol. **3** (1994), pp. 101–130.

[9] Durán, F., M. Roldán and A. Vallecillo, *Invariant-driven strategies for Maude*, ENTCS **124** (2005), pp. 17–28, Proc. 4th Intl. Workshop on Reduction Strategies in Rewriting and Programming (WRS 2004).

[10] Goguen, J. A., T. Winkler, J. Meseguer, K. Futatsugi and J.-P. Jouannaud, *Introducing OBJ*, in: J. A. Goguen and G. Malcolm, editors, *Software Engineering with OBJ: Algebraic Specification in Action*, Kluwer, 2000 pp. 3–167.

[11] Hoare, C. A. R., "Communicating Sequential Processes," Prentice Hall, 1985.

[12] International Telecommunication Union, *Open Distributed Processing - Reference Model parts 1–4*, Technical report, ISO/IEC, Geneva (1995).

[13] Johnsen, E. B. and O. Owe, *A compositional formalism for object viewpoints*, in: B. Jacobs and A. Rensink, editors, *Proc. 5th Intl. Conf. on Formal Methods for Open Object-Based Distributed Systems (FMOODS'02)* (2002), pp. 45–60.

[14] Johnsen, E. B. and O. Owe, *Object-oriented specification and open distributed systems*, in: O. Owe, S. Krogdahl and T. Lyche, editors, *From Object-Orientation to Formal Methods: Essays in Memory of Ole-Johan Dahl*, LNCS **2635**, Springer-Verlag, 2004 pp. 137–164.

[15] Johnsen, E. B., O. Owe and E. W. Axelsen, *A run-time environment for concurrent objects with asynchronous method calls*, in: N. Martí-Oliet, editor, *5th Intl. Workshop on Rewriting Logic and its Applications (WRLA'04)*, ENTCS **117** (2005), pp. 375–392.

[16] Jones, C. B., "Development Methods for Computer Programmes Including a Notion of Interference," Ph.D. thesis, Oxford University, UK (l981).

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

[18] Meseguer, J. and G. Roşu, *Rewriting logic semantics: From language specifications to formal analysis tools*, in: D. A. Basin and M. Rusinowitch, editors, *Proc. 2nd Intl. Joint Conf. on Automated Reasoning (IJCAR 2004)*, LNCS **3097** (2004), pp. 1–44.

[19] Pretschner, A., H. Lötzbeyer and J. Philipps, *Model based testing in incremental system development*, Journal of Systems and Software **70** (2004), pp. 315–329.

[20] Roşu, G. and K. Havelund, *Rewriting-based techniques for runtime verification*, Journal of Automated Software Engineering **12** (2005), pp. 151–197.

[21] Rusu, V., H. Marchand, V. Tschaen, T. Jéron and B. Jeannet, *From safety verification to safety testing*, in: R. Groz and R. M. Hierons, editors, *16th IFIP Intl. Conf. on Testing of Communicating Systems (TestCom 2004)*, LNCS **2978** (2004), pp. 160–176.

[22] Tyler, B. and N. Soundarajan, *Black-box testing of grey-box behavior*, in: A. Petrenko and A. Ulrich, editors, *3rd Intl. Workshop on Formal Approaches to Testing of Software (FATES 2003)*, LNCS **2931** (2004), pp. 1–14.

[23] Van Aertryck, L., M. Benveniste and D. Le Metayer, *CASTING : A formally based software test generation method*, in: *Intl. Conf. on Formal Engineering Methods (ICFEM'97)* (1997), pp. 101–110.

[24] Venkatasubramanian, N. and C. L. Talcott, *Reasoning about meta level activities in open distributed systems*, in: *Proc. 14th Symp. on Principles of Distributed Computing (PODC '95)* (1995), pp. 144–152.

[25] Walter, T., I. Schieferdecker and J. Grabowski, *Test Architectures for Distributed Systems - State of the Art and Beyond*, in: A. Petrenko and N. Yevtushenko, editors, *11th Intl. Workshop on Testing Communicating Systems (IWTCS)*, IFIP Conference Proceedings **131** (1998), pp. 149–174.