Toward Reflective Application Testing in Open Environments

Eyvind W. Axelsen, Einar Broch Johnsen, and Olaf Owe
Department of Informatics, University of Oslo

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 uncontrollable ways, as other applications may arrive, change, or disappear. In order to test the behavior of components in such environments, it is necessary to build a testing environment which reflects this highly unpredictable behavior. To avoid over-specification of environment components, we use the observable communication history to abstractly reflect the state of communicating components. Rewriting logic has been used to capture many different systems of concurrency and communication in an executable manner. In this paper, we show how rewriting logic models can be extended with observable communication histories in a transparent way and suggest using this extension to capture a form of assumption guarantee specification based testing of components in open environments.

1 Introduction

The aim of this paper is to suggest an application of rewriting logic [18] to test the behavior of software units in *open environments* such as the Internet. An open environment is an environment in which various other software units exist, and no specific knowledge about these units may be assumed. A software unit in this setting may be a distributed application in an open distributed system, but also an off-the-shelf component which should behave properly in a variety of environments. To keep the exposition simple, we will model software units by *objects* communicating by means of *message passing* in this paper, keeping in mind that the approach may in principle be extended to "real components" or distributed applications.

It is a major challenge to predict the behavior of objects evolving in open environments, in order to ensure and maintain behavioral properties such as safety, availability, quality of service, various forms of fault tolerance, etc. Formal approaches to this challenge include methods for verification, by means of e.g. assertion systems such as Hoare logic, type checking, and model checking. A disadvantage of these approaches is that they generally depend on knowledge of the implementation details of the systems they consider. In contrast, approaches based on testing create an artificial environment in which the object can be subjected to controlled test runs. In contrast to verification methods, testing cannot generally ensure that components are well-behaved at all times, but may still give revealing insights into a component's behavior. This paper takes a testing approach to object analysis. The goal of the paper is to show how open environments can be mimicked by underspecified formal descriptions based on *observable behavior* in order to test object behavior.

Testing is done in an executable platform defined using *rewriting logic* and the Maude system [5, 18]. Rewriting logic can naturally express and combine many different models of communication and concurrency. Further, rewriting logic is *reflective* [3, 6] 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 *meta-level*. 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 [16]. 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 [23]. In this paper, we transparently extend an executable system model with its history of observable communications [7, 12] at the meta-level, and define execution strategies at the meta-level which are influenced by requirements on the history. This paper gives an overview, further technical details of the implementation of this work in Maude may be found in [1].

Paper overview: Section 2 briefly reviews a notion of behavioral interface and introduces the example of the paper. Section 3 introduces rewriting logic and Maude. Section 4 provides an executable Maude specification of the running example. Section 5 explains how histories are introduced into the Maude model. Section 6 considers testing of observable behavior. Section 7 provides an executable *abstract* Maude specification of the running example. Section 8 combines two meta-level strategies to create an open testing environment. Section 9 concludes the paper.

2 Specifications of Observable Behavior

In the open distributed setting, objects in the environment may come from third-party manufacturers, and their implementation details may be unavailable for various reasons. Reasoning about the overall system behavior should be based on abstract specifications of system components. Specifications in terms of observable behavior seem particularly attractive, assuming that components have behavioral interfaces that describe their use. A component may have many interfaces corresponding to different behavioral requirements.

Interaction histories. In a distributed environment, object behavior may be given in terms of an assumption guarantee specification [17]. The assumption is a requirement on the behavior of the objects in the environment. As customary in the assumption-guarantee paradigm, the guaranteed invariant need only hold when the environment respects the assumption. In our setting, the paradigm is adjusted to deal with input and output aspects of communicating systems. An object's observable history, i.e. the trace of all communication events between the object and its environment, represents an abstract view of its state, available for reasoning about past and present behavior. An object's behavior may be determined by its communication history up to present time, and a specification of its behavior may be given as a predicate on finite traces. The approach emphasizes mathematically intuitive concepts such as generator inductive function definitions and finite sequences, avoiding fix-point semantics and infinite traces [15].

Behavioral interfaces. In order to specify object behavior in a generic way, we introduce interfaces which may be associated with objects. Interfaces that contain semantic requirements can be understood as *behavioral interfaces*. An interface can be

implemented by different classes and a class can implement different interfaces. If a class implements an interface, all the objects of the class must behave according to the semantic requirements of the interface, which describe observable behavior in terms of possible communication histories. For further technical details and discussion the reader is referred to [13, 15]. We shall here proceed by an example.

Example. Consider the well-known dining philosophers example [9] with N philosophers seated around a table. They are thinking deeply, but may occasionally need to eat from a common resource. Each philosopher is equipped with one chopstick, but in order to eat two chopsticks are needed. Hence, a philosopher may request and return its right-hand neighbor's chopstick, and lend its stick to its left-hand neighbor in response to a request. These are the possible philosopher operations. We assume that philosophers are initially thinking, but at some point they may request their neighbor's chopstick, and lend their chopstick to their left-hand neighbor. A philosopher which controls two chopsticks may eat, return the requested chopstick, and resume thinking.

We specify an interface *Phil*, describing observable philosopher behavior, including eating, thinking, and communication concerning the exchange of sticks. Let X and Y be variables ranging over possible philosophers. Semantically, we represent an interaction by a triple $\langle X, Y, M\rangle$, where X is the caller, Y the callee, and M a message. Remark that *think* and *eat* are internal interactions, i.e. $X = Y$, while *requestStick*, *returnStick* and *lendStick* are external, i.e. $X \neq Y$. The behavior of a philosopher is formulated in terms of acceptable communication histories, which observationally reflect the state of a philosopher. The following specification ensures that eating requires two sticks, thinking no borrowed stick, and that the borrowing and lending of sticks are done properly:

interface *Phil*
begin
  opr *think*
  opr *eat*
  opr *requestStick* — request stick from neighbor
  opr *returnStick* — return stick to neighbor
  opr *lendStick* — lend stick to neighbor
 inv $AccBeh(this, \mathcal{H})$
where $AccBeh(X, \varepsilon) \equiv \mathsf{true}$
$AccBeh(X, \mathcal{H} \vdash \langle X, Y, think\rangle) \equiv AccBeh(X, \mathcal{H})) \wedge \neg otherStick?(X, \mathcal{H})$
$AccBeh(X, \mathcal{H} \vdash \langle X, Y, eat\rangle) \equiv AccBeh(X, \mathcal{H}) \wedge myStick?(X, \mathcal{H}) \wedge otherStick?(X, \mathcal{H})$
$AccBeh(X, \mathcal{H} \vdash \langle X, Y, requestStick\rangle) \equiv AccBeh(X, \mathcal{H})) \wedge \neg otherStick?(X, \mathcal{H})$
$AccBeh(X, \mathcal{H} \vdash \langle X, Y, returnStick\rangle) \equiv AccBeh(X, \mathcal{H}) \wedge otherStick?(X, \mathcal{H}) \wedge$
$\mathcal{H}/\{\langle X, Y, lendStick\rangle, \langle Y, X, returnStick\rangle\}$ **ew** *lendStick*
$AccBeh(X, \mathcal{H} \vdash \langle X, Y, lendStick\rangle) \equiv AccBeh(X, \mathcal{H}) \wedge myStick?(X, \mathcal{H}) \wedge$
$\mathcal{H}/\{\langle Y, X, requestStick\rangle, \langle X, Y, lendStick\rangle\}$ **ew** *requestStick*
$AccBeh(X, \mathcal{H} \vdash \langle X', Y, M\rangle) \equiv AccBeh(X, \mathcal{H})$ **if** $(X \neq X')$

$myStick?(X, \mathcal{H}) \equiv \neg(\mathcal{H}/\{\langle X, \_, lendStick\rangle, \langle \_, X, returnStick\rangle\}$ **ew** *lendStick*$)$
$otherStick?(X, \mathcal{H}) \equiv\ (\mathcal{H}/\{\langle \_, X, lendStick\rangle, \langle X, \_, returnStick\rangle\}$ **ew** *lendStick*$)$
end

In the definition of *AccBeh*, each line corresponds to a possible generator case. The free variables in each equation have an implicit universal quantifier, reminiscent of for instance ML, letting _ match any term. The empty sequence is denoted ε, and **ew** denotes "ends

with". The restriction of a history h to a set S of messages is denoted h/S. Note that the specification is underspecified with regard to philosopher behavior. In particular, we do not know if a philosopher will lend its chopstick to its neighbor when hungry (i.e. has made a *requestStick*), and if a hungry philosopher will make many stick requests.

3 Rewriting Logic and Maude

This section gives a brief introduction to rewriting logic [18] and Maude [5]. 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 [5, 18], including Petri nets, CCS, Actors, and Unity, as well as the ODP computational model [19] and real-time systems [20]. RL is also used to define the operational semantics of the Creol language [14, 16], and additionally offers its own model of object orientation [5].

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 [11] style. When modeling computational systems, configurations may include the local system states, where different parts of the system are 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 [5].

RL extends algebraic specification techniques with transition rules: The dynamic behavior of a system is captured by rewrite rules, supplementing the equations which define 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. If several rules can be applied to distinct subconfigurations, they can be executed in a concurrent rewrite step. Consequently, concurrency is implicit in RL.

Conditional rewrite rules are allowed, where the condition can be formulated as a conjunction of rewrites and equations which must hold for the main rule to apply:

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

Rules in RL may be formulated at a high level of abstraction, closely resembling a compositional operational semantics. In fact, modular structural operational semantics can be uniformly mapped into RL specifications [8].

Reflection and the Maude Meta-Level. 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 $\mathcal{U}$ [4].

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 meta-level. Using this notation, we have the equivalence

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

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.

Meta-level rewrite rules may be used to select which rule from $\mathcal{R}$ to apply to which subterm of C. This is done by a sequential interpreter function which takes as arguments a finitely presented rewrite theory $\mathcal{R}$, a term C, and a deterministic strategy S. Details on the theory and use of reflection in rewriting logic and Maude may be found in [3, 4, 6].

4 A Maude Model of the Dining Philosophers

The dining philosophers of Section 2 are now implemented in Maude. In order to define the *synchronization constraints*, we introduce internal attributes in the philosopher objects to determine their ability to perform the different possible actions. Let C and D be variables ranging over the sort *Cid* of *concrete* philosopher identifiers, a subsort of *Phil*. A concrete philosopher with internal structure may be defined as follows:

$$\langle C : Cid \mid state : \_, myStick : \_, nbrStick : \_, leftNbr : \_, rightNbr : \_ \rangle$$

The philosophers interact asynchronously by passing messages to each other, as well as by sending synchronous messages to themselves representing internal actions.

A message consists of an envelope with a sender and a receiver and the actual message content (or action) represented by a *quoted identifier*. In Maude, messages are conventionally defined as follows:

$$msg\_\ from\_\ to\_$$

We are now able to define synchronization constraints directly in terms of a philosopher's internal state. A *hungry* philosopher may attempt to gain control of two chopsticks in order to eat. A *fed* philosopher may think. Hence, the *state* may be either *hungry* or *fed*. When a philosopher is hungry, it will attempt to acquire chopsticks. When it controls two chopsticks, it may eat, release the chopsticks, and resume thinking. This can be expressed by the following rewrite rules, ignoring attributes that are not needed for synchronization in the style of Full-Maude [5]:

rl $[think] : \langle C : Cid \mid state : fed \rangle \longrightarrow \langle C : Cid \mid state : fed \rangle$ $(msg\ 'think\ from\ C\ to\ C)$.

rl $[eat]$:
$\langle C : Cid \mid state : hungry, myStick : yes, nbrStick : yes \rangle$
$\longrightarrow \langle C : Cid \mid state : hungry, myStick : yes, nbrStick : yes \rangle$ $(msg\ 'eat\ from\ C\ to\ C)$.

rl $[requestStick]$:
$\langle C : Cid \mid state : fed, myStick : yes, nbrStick : no, rightNbr : D \rangle$
$\longrightarrow \langle C : Cid \mid state : hungry, myStick : yes, nbrStick : requested, rightNbr : D \rangle$
$(msg\ 'requestStick\ from\ C\ to\ D)$.

rl $[returnStick]$:
$\langle C : Cid \mid state : hungry, myStick : yes, nbrStick : yes, rightNbr : D \rangle$
$\longrightarrow \langle C : Cid \mid state : fed, myStick : yes, nbrStick : no, rightNbr : D \rangle$
$(msg\ 'returnStick\ from\ C\ to\ D)$.

rl $[lendStick]$:
$\langle C : Cid \mid state : fed, myStick : yes \rangle$ $(msg\ 'requestStick\ from\ D\ to\ C)$
$\longrightarrow \langle C : Cid \mid state : fed, myStick : no \rangle$ $(msg\ 'lendStick\ from\ C\ to\ D)$.

rl [*recieveRequestedStick*] :
$\langle C : Cid \,|\, nbrStick : requested, rightNbr : D \rangle$ (*msg 'lendStick from D to C*)
$\longrightarrow \langle C : Cid \,|\, nbrStick : yes, rightNbr : D \rangle$

rl [*recieveReturnedStick*] :
$\langle C : Cid \,|\, myStick : no, rightNbr : D \rangle$ (*msg 'returnStick from D to C*)
$\longrightarrow \langle C : Cid \,|\, myStick : yes, rightNbr : D \rangle$

Note that in this specification, a hungry philosopher will not render its chopstick.

5 Extending the Model with Histories

In the executable model, the messages corresponding to the philosophers' actions can be recorded in a *communication history*. In this section we will look at how to utilize Maude's meta-level capabilities during execution of the specified model to record the history in a *transparent* way, i.e. leaving the original specification unchanged.

In order to execute a specification at the meta-level, we develop a custom *strategy*, i.e. rewrite rules which apply to the meta-representation of the model. This strategy makes use of an object *Engine* to store the information we need in order to control consecutive meta-level rewrites. The engine object is defined as follows:

$$\langle E : Engine \,|\, curTerm : \_, curModule : \_, labels : \_, failedRules : \_ \rangle$$

This object contains several attributes, whose values are set at run-time; *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 the module *curModule*, and *failedRules* contains a list of rule labels for rules that could not be applied to *curTerm*.

The communication history is kept in a Maude object, *History*, which is distinct from the objects of the object-level model and is defined as follows:

$$\langle H : History \,|\, h : \_ \rangle$$

The only attribute of this object, *h*, will contain the actual communication history of the system in the form of a message list recorded at runtime.

By defining a custom strategy we gain control over when a meta-level rewrite is performed, and hence we are able to inspect the current state in-between rewrites. This is the key point that enables us to record a communication history while executing a specification, since we can now check whether the application of a given rewrite rule to a given configuration results in a new message being sent, by comparing the meta-level representations of the configuration before and after the rule application, respectively.

Our strategy is implemented by a conditional rewrite rule *exec*, defined below. The actual term rewriting is performed by Maude's built-in *descent function metaXapply*($\overline{\mathcal{R}}$, $\bar{t}$, $\bar{l}$, σ, n, b, m). This (partial) function provides fine-grained control over rewriting at the meta-level, allowing us to specify which rewrite rule ($\bar{l}$) is to be applied to which term ($\bar{t}$), and at which position (m) within the term the rule is to be applied if there are several possibilities. For a more detailed description of this function, see [5]

The function *metaXapply* returns a four-tuple consisting of the resulting term, the type of the term, a substitution and a context, and we use a function *getTerm* to extract the rewritten term from this tuple. Note that whitespace in Maude denotes list concatenation: If L is a label and *LABELS* is a list of labels, then L *LABELS* is a non-empty list of labels.

```
crl [exec] :
    ⟨E : Engine | curTerm : T, curModule : MOD, labels : L LABELS, failedRules : FR⟩
    ⟨H : History | h : ML⟩
⟶
    if metaXapply([MOD], T, L, none, 0, unbounded, 0) =/= failure then
        ⟨E : Engine | curTerm : getTerm(metaXapply([MOD],T, L, none, 0, unbounded, 0)),
          curModule : MOD, labels : LABELS L, failedRules : nil⟩
        ⟨H : History | h : ML + getNewMessages(T, getTerm(metaXapply([MOD],
          T, L, none, 0, unbounded, 0)), MOD, ML)⟩

    else
        ⟨E : Engine | curTerm : T, curModule : MOD, labels : LABELS L, failedRules : FR⟩
        ⟨H : History | h : ML⟩
    fi
if length(FR) < length(L LABELS) .
```

This strategy applies rules from the *labels* list in a round-robin fashion to the meta-level configuration in *curTerm*. (Remark that we may define other strategies for rule selection than round-robin. In [16], we have shown how to extend a similar strategy such that the rules are selected randomly using a pseudo-random number generator.) In the event that no rule is applicable, the execution will terminate.

The auxiliary function *getNewMessages* compares the term T with the term resulting from applying (with *metaXapply*) 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 we can add them to the history in an arbitrary order.

Using the strategy defined above, we may execute a Maude specification, such as the one for the dining philosophers problem introduced in Section 4, and record any messages that are being sent, without changing anything in the original specification. Hence, the analysis capabilities obtained by recording the history may be added to the model when needed and removed after sufficient analysis.

6 A Strategy for Testing Observable Behavior

The communication history that is built at runtime when applying the rewrite strategy introduced in Section 5, can be used as input to a *test oracle*, blocking execution if a given Maude specification violates a given behavioral specification. This can be achieved by extending the strategy with functionality for checking whether a given rule application will lead to an illegal state, as specified by a given predicate. Taking the observational approach, we consider predicates on communication histories. In order to obtain a compositional system, the predicate on the global history will be the conjunction of a number of behavioral interfaces, associated with different objects. Behavioral specifications for specific object-level objects are represented by trace predicates on the global history, restricted to an appropriate subset of possible communication events. The overall picture is illustrated by Figure 1: Let $\mathcal{R}$ be the object-level set of rewrite rules and $\mathcal{C}$ a system state. A meta-level strategy $\mathcal{S}$ controls how the choice of rule is made for application at the object level. The strategy is here parameterized by a predicate on communication histories. For the testing strategy of this section, a *fail-stop* strategy is defined, which blocks further execution once the system attempts to violate the predicate P on the global history. If execution is blocked by the strategy, the recorded history

	Rule set:	Configuration:
Meta-level rewrite system:	$\mathcal{S}_{\textit{fail-stop}}(P(h))$	$\overline{\mathcal{R}}, \overline{\mathcal{C}}, \langle H : \textit{History} \mid h : \_ \rangle$
	$\downarrow$ Control	$\uparrow$ History logger
Object level rewrite system:	$\mathcal{R}$	$\mathcal{C}$

Figure 1: Reflective testing of observable behavior.

provides an error trace for the system run, describing how the specification was violated.

In order to implement *fail-stop* testing in Maude, we first define a constant H of a new sort *History*, which we will use as a placeholder for the actual communication history $\mathcal{H}$ (which will be recorded during execution, and hence is not available at the time of specification) in the specification of our predicates. Furthermore, we define a sort *Pred*, the sort of predicates on communication histories. For the running example, acceptable behavior for a system of philosophers behaving according to the behavioral interface defined in Section 2, can then be expressed by the Maude operator

$$\textbf{op}\ \textit{AccBeh}\ :\ \textit{History} \rightarrow \textit{Pred}\ .$$

During execution, the predicate needs to be checked between each rewrite step. For this purpose, we introduce the function $\textit{CheckPredicate} : \textit{Pred} \times \textit{MsgList} \rightarrow \textit{Bool}$. This function will be called by the strategy from Section 5, and will parse the predicate specification, call auxiliary predicate checking functions and return a boolean value indicating whether the message list is in compliance with the predicate or not.

Returning to the *AccBeh* predicate, the case in which *think* is the last message in the history can be specified equationally in Maude as follows:

eq *AccBeh(ML + msg 'think from X to Y)* $=$ *AccBeh(ML)* **and not** $\textit{otherStick?}(X, ML)$.
eq $\textit{otherStick?}(X, nil) = true$.
eq $\textit{otherStick?}(X, ML + \textit{msg 'returnStick from X to Y}) = false$.
eq $\textit{otherStick?}(X, ML + \textit{msg 'lendStick from Y to X}) = true$.
eq $\textit{otherStick?}(X, ML + M) = \textit{otherStick?}(X, ML)$ [**otherwise**].

Here, X and Y are variables of sort *Phil*, M is a message of sort *Msg*, and *ML* is the communication history in the form of a message list as recorded by our strategy during execution. The last equation for *otherStick?* is chosen only when the others do not match. Note that 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*.

If the communication history after a given rewrite is not in compliance with the predicate, the execution will terminate and a "failure object" will be inserted into the configuration to indicate what went wrong. For more details on the implementation of the predicate check, see [1].

7 An Executable Abstract Prototype of Dining Philosophers

We now define a prototype philosopher model in Maude. The idea is to postpone design decisions concerning the concrete internal structure and rather define the behavior of the philosopher in terms of restrictions on the history of its observable actions. Let A and B

be variables ranging over object identifiers of sort *Aid* for *abstract* philosophers. Define a sort *AbsPhil* with terms $\langle A : Aid \mid leftNbr : \_,\ rightNbr : \_ \rangle$. Except for philosopher identities and their left- and right-hand side neighbors, we completely ignore any internal structure in the philosopher objects. Consequently, synchronization constraints cannot be expressed in terms of the internal state. The possible actions of the abstract philosophers are captured by the following rewrite rules:

$$\textbf{rl}\ [think] :\ \langle A : Aid \mid \rangle \longrightarrow \langle A : Aid \mid \rangle\ (msg\ 'think\ from\ A\ to\ A)\ .$$

$$\textbf{rl}\ [eat] :\ \langle A : Aid \mid \rangle \longrightarrow \langle A : Aid \mid \rangle\ (msg\ 'eat\ from\ A\ to\ A)\ .$$

$$\begin{array}{l}\textbf{rl}\ [requestStick] : \\ \quad \langle A : Aid \mid rightNbr : B \rangle \\ \longrightarrow \langle A : Aid \mid rightNbr : B \rangle\ (msg\ 'requestStick\ from\ A\ to\ B)\ .\end{array}$$

$$\begin{array}{l}\textbf{rl}\ [returnStick] : \\ \quad \langle A : Aid \mid rightNbr : B \rangle \\ \longrightarrow \langle A : Aid \mid rightNbr : B \rangle\ (msg\ 'returnStick\ from\ A\ to\ B)\ .\end{array}$$

$$\begin{array}{l}\textbf{rl}\ [lendStick] : \\ \quad \langle A : Aid \mid leftNbr : B \rangle \longrightarrow \\ \quad \langle A : Aid \mid leftNbr : B \rangle\ (msg\ 'lendStick\ from\ A\ to\ B)\ .\end{array}$$

These rules do not express any synchronization constraints on the interactions, only which philosophers may interact. Also note, that rules for receiving messages are no longer needed, since no internal state change takes place in the abstract philosopher objects. Instead, a simple consumption rule can be used to remove messages from the configuration:

$$\textbf{rl}\ [consumeMsg] :\ \langle A : Aid \mid \rangle (msg\ M\ from\ B\ to\ A) \longrightarrow \langle A : Aid \mid \rangle\ .$$

8 Simulating Open Environments by Behavioral Interfaces

In an open environment, objects may be created and destroyed dynamically during execution. With our abstract philosopher specification, this can be modeled by the following rewrite rules:

$$\begin{array}{l}\textbf{rl}\ [create] : \\ \quad \langle A : Phil \mid rightNbr : B \rangle\ \langle B : Phil \mid leftNbr : A \rangle \\ \longrightarrow \langle A : Phil \mid rightNbr : A + B \rangle\ \langle B : Phil \mid leftNbr : A + B \rangle \\ \quad \langle A + B : Aid \mid leftNbr : A, rightNbr : B \rangle \\ .\end{array}$$

$$\begin{array}{l}\textbf{rl}\ [destroy] : \\ \quad \langle A : Phil \mid rightNbr : B \rangle\ \langle B : Aid \mid leftNbr : A, rightNbr : C \rangle\ \langle C : Phil \mid leftNbr : B \rangle \\ \longrightarrow \langle A : Phil \mid rightNbr : C \rangle\ \langle C : Phil \mid leftNbr : A \rangle\ .\end{array}$$

In the *create* rule, the new abstract philosopher object is inserted between two existing (abstract or concrete) philosopher objects. The new philosopher will have the concatenation of the existing objects' identifiers as its identifier. In the *destroy* rule, an abstract philosopher object in-between two other philosopher objects is deleted, and the remaining philosopher objects set their *leftNbr* and *rightNbr* properties accordingly.

Using our abstract dining philosopher specification with the rules introduced above, we can simulate an open environment the behavior of which is exclusively defined by a behavioral specification in the form of predicates at the meta-level. As mentioned in Section 5, the meta-level can be used to define a strategy which stops the execution of a

	Rule set:	Configuration:
Meta-level:	$\mathcal{S}_{force}(P_1(h/\alpha_1)) \wedge \mathcal{S}_{fail\text{-}stop}(P_2(h/\alpha_2))$	$\overline{\mathcal{R}_1} \cup \overline{\mathcal{R}_2}, (\overline{\mathcal{C}_1}\ \overline{\mathcal{C}_2}), \langle H : History \mid h : \_ \rangle$
	$\downarrow$ Control	$\uparrow$ History logger
Object level:	$\mathcal{R}_1 \cup \mathcal{R}_2$	$\mathcal{C}_1\ \mathcal{C}_2$

Figure 2: Reflective testing of observable behavior in open environments.

specification if a given rule application violates the predicate. However, when defining an abstract environment where every rule is applicable at any time (because there is no synchronization code in the objects), this strategy is not desirable here. Instead, we want to *force* the abstract specification to behave in compliance with the predicate. This can be achieved by a similar strategy, using the mechanisms introduced in Section 6. However, where the *fail-stop* strategy halts the execution when the application of an enabled rule will break the predicate, the new *force* strategy will try another enabled rule from the *labels* list of the *Engine* object instead. Execution will first terminate when no rule can be applied without violating the predicate.

The abstract environment specification can now be used as a "testbed" for a number of actual programmed components, like the philosophers from Section 2. Let $\mathcal{R}_1$ be an object-level set of rewrite rules which may be applied to a system configuration $\mathcal{C}_1$, the system state of abstract objects (the open environment), and let $\mathcal{R}_2$ be the object-level set of rewrite rules which may be applied to the concrete objects in a system configuration $\mathcal{C}_2$ (the given components, with synchronization constraints on the internal state). Let α_1 and α_2 be messages associated with the objects of C_1 and C_2, respectively. Messages may be exchanged freely between all objects, so the two sets are not disjoint. Let P_1 and P_2 be predicates observationally specifying the environment and actual components, respectively. The meta-level strategy $\mathcal{S}_{force}$ restricts rule application from $\mathcal{R}_1$ to acceptable environment behavior. This provides an abstract, open environment which may behave in any way that does not violate the specification P_1. We here combine two meta-level strategies which react differently to the violation of predicates: *force* will restrict rule application so that the communication history conforms to the predicate, and *fail-stop* will halt the execution and produce an error object if the predicate is attempted violated. By specifying one predicate that spans only messages from the programmed object, and one that spans all objects, and by checking the former in *fail-stop* mode and the latter in *force* mode, we can test whether our programmed component execute correctly provided that the environment does so. This scenario is illustrated by Figure 2.

To illustrate this scheme by the running example of this paper, we may consider a system of dining philosophers, where some philosophers are implemented, i.e. belong to the concrete philosophers defined in Section 4. We want to test that these concrete objects behave according to the requirements of the *Phil* interface by means of the *fail-stop* strategy. To create an environments in which they interact with other, abstract philosophers, we introduce the abstract philosophers of Section 7. In order to make the abstract philosophers behave according to the same *Phil* interface, we use the *force* strategy. In Figure 2, the *Phil* interface will be used as both predicates P_1 and P_2.

Alternatively, if the *Phil* interface was given in a rely/guarantee style with minimal requirements to the environment, the latter could be used in force mode and the guarantee part in fail-stop mode.

9 Conclusion and Future Work

This paper shows how abstract specifications of dynamic environments may be captured very naturally in a rewriting logic model extended with behavioral interfaces. Due to the reflective character of rewriting logic, supported by Maude, it is possible to define execution strategies at the meta-level. In this paper, we have used this facility to test whether an executable model is well behaved with respect to a number of requirements on the observable behavior of the model, defined as behavioral interfaces.

Further, we show how meta-level strategies may be used to execute a prototype model defined by its observable behavior, without deciding on its implementation details. Combining these meta-level strategies, we obtain abstract testing environments for models of components or distributed applications, in which the environment is unspecified but subjected to certain minimal observational requirements. Previous approaches to history-based testing, e.g. [10, 21], and automata-based approaches, e.g. [2, 22], require specific test cases to be defined. In contrast, our approach uses *random testing* and assumption guarantee specifications to define *open environments*. Further, the environment and the test oracle are defined within the same formalism. Methods to generate additional restrictions on assumptions to obtain reasonable coverage remain to be addressed.

Some experiments with socket extensions to Maude suggest that it is possible to employ Maude processes as demonstrated in this paper to act as a testing environment and to simulate an open environment for actual components communicating by means of a predefined set of messages with the Maude process via sockets. However, future work in this vein remains.

References

[1] E. W. Axelsen. A meta-level framework for recording and utilizing communication histories in Maude. Master's thesis, Department of Informatics, University of Oslo, Norway, Aug. 2004.

[2] S. Barbey, D. Buchs, and C. Péraire. A theory of specification-based testing for object-oriented software. In *Proceedings of EDCC2 (European Dependable Computing Conf.)*, LNCS 1150, pages 303–320. Springer, 1996.

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

[4] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Marí-Oliet, and J. Meseguer. Meta-level computation in Maude. In *2nd Intl. Workshop on Rewriting Logic and its Applications (WRLA'98)*, volume 15 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1998.

[5] M. Clavel, 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*, 285:187–243, Aug. 2002.

[6] M. Clavel and J. Meseguer. Reflection in conditional rewriting logic. *Theoretical Computer Science*, 285:245–288, Aug. 2002.

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

[8] C. de Oliveira Braga. *Rewriting Logic as a Semantic Framework for Modular Structural Operational Semantics*. PhD thesis, Pontifícia Universidade Católica do Rio de Janeiro, 2001.

[9] E. W. Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976.

[10] R.-K. Doong and P. G. Frankl. The ASTOOT approach to testing object-oriented programs. *ACM Trans. Softw. Eng. Methodol.*, 3(2):101–130, 1994.

[11] J. Goguen, T. Winkler, J. Meseguer, K. Futatsugi, and J.-P. Jouannaud. Introducing OBJ. In J. Goguen, editor, *Applications of Algebraic Specification using OBJ*. Cambridge, 1993.

[12] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[13] E. B. Johnsen 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)*, pages 45–60. Kluwer, Mar. 2002.

[14] E. B. Johnsen and O. Owe. An asynchronous communication model for distributed concurrent objects. In *Proc. 2nd IEEE Intl. Conf. on Software Engineering and Formal Methods (SEFM'04)*, pages 188–197. IEEE CS Press, Sept. 2004.

[15] E. B. Johnsen 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, pages 137–164. Springer, 2004.

[16] E. B. Johnsen, O. Owe, and E. W. Axelsen. A run-time environment for concurrent objects with asynchronous methods calls. In *Proc. 5th Intl. Workshop on Rewriting Logic and its Applications (WRLA'04)*, To appear in ENTCS. Elsevier, 2004.

[17] C. B. Jones. *Development Methods for Computer Programmes Including a Notion of Interference*. PhD thesis, Oxford University, UK, June l981.

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

[19] E. Najm and J.-B. Stefani. A formal semantics for the ODP computational model. *Computer Networks and ISDN Systems*, 27:1305–1329, 1995.

[20] P. C. Ölveczky and J. Meseguer. Specification of real-time and hybrid systems in rewriting logic. *Theoretical Computer Science*, 285:359–405, 2002.

[21] B. Tyler 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, pages 1–14. Springer, 2004.

[22] L. Van Aertryck, M. Benveniste, and D. Le Metayer. CASTING : A formally based software test generation method. In *ICFEM'97, First IEEE Intl. Conf. on Formal Engineering Methods*, pages 101–110, Nov. 1997. IEEE CS Press.

[23] N. Venkatasubramanian and C. L. Talcott. Reasoning about meta level activities in open distributed systems. In *Proc. the 14th Annual ACM Symp. on Principles of Distributed Computing (PODC '95)*, pages 144–152, Aug. 1995. ACM Press.