Model Testing Asynchronously Communicating Objects using Modulo AC Rewriting

Olaf Owe, Martin Steffen, and Arild B. Torjusen

Department of Computer Science, University of Oslo, Norway

Abstract

Testing and verification of asynchronously communicating objects in open environments are challenging due to non-determinism. We explore a formal approach for black-box testing by proposing an interface specification language that gives an assumption-commitment style description of an object's behavior. The approach is applied to Creol objects. Creol is a high-level, object-oriented modelling language, hence we do model-based testing of behavioral models. The testing is done by synchronising execution of a specification and the component under test. Due to the asynchronous nature of communication, testing should be done up-to observational equivalence. This leads to a large increase in the reachable state space for the test cases. We reduce the state space by using facilities for rewriting modulo AC (associativity and commutativity) built into the rewriting logic system Maude, and explore the state space by breadth first search. We present experimental results that show the usefulness of this approach.

Keywords: Testing and verification, asynchronous method calls, active objects, rewriting logic, formal semantics.

1 Introduction

Systematic testing is indispensable to assure reliability and quality of software and systems. Hosts of different testing approaches and frameworks have been proposed and put to (good) use over the years. Formal methods and program language theory have proven valuable to render testing practice a more formal, systematic discipline (cf. e.g. [16,2]). Formal approaches to testing have gained momentum in recent years, as for instance witnessed by the trend towards model-based testing [12,4]. In previous work [19] we presented a formal approach for black-box specification-based testing of asynchronously communicating components in open environments together with an implementation of a testing framework. In this paper we show how to extend the approach to *verification* of components and present experimental results that show the usefulness of our approach.

⋆ Part of this work has been supported by the EU-project IST-33826 *Credo: Modeling and analysis of evolutionary structures for distributed services*, *HATS: Highly Adaptable and Trustworthy Software using Formal Methods* (`http://www.hats-project.eu`), and the German-Norwegian DAAD-NWO exchange project *Avabi* (Automated validation for behavioral interfaces of asynchronous active objects).

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

We do this in the context of Creol [11,27], a high-level, object-oriented modelling language for distributed systems. Object-orientation is a natural choice, as object modelling is the fundamental approach to open distributed systems as recommended by RM-ODP [24]. For such systems an *asynchronous* communication model is advantageous as it decouples caller and callee thus avoiding unnecessary waiting for method returns. On the downside, asynchronicity makes verifying and testing models more challenging. In an asynchronous system, communication delays due to the network or to queuing may lead to message overtaking and the resulting non-determinism leads to a state space explosion.

It is generally accepted that the way to tackle complex systems is to "divide-and-conquer", i.e., consider components interacting with their environment. Abstracting from internal executions, the black-box behavior of Creol components is given by interactions at their *interface*. We use a concise language over communication labels to specify components and the expected behavior of a component is given as a set of traces at the interface. Both input and output interactions are specified but play quite different roles. As input events are not under the control of the object, input is considered as assumptions about the environment whereas output describes commitments of the object. This separation of concerns between interaction under the control of the component and coming from the environment leads to an assumption-commitment style specification of a component's behavior by defining the valid observable output behavior, assuming a certain scheduling.

For input interactions, we ensure that the specified assumptions on the environment are fulfilled by *scheduling* the incoming calls in the order specified, while for output events, which are controlled by the component, we *test* that the events occur as specified. Scheduling and testing of a component are done by synchronizing the component's execution with the specification. As a result, the scheduling is enforced in the execution of the component and the actual outgoing interactions from the component are tested against the output events in the specification. This gives a framework for testing whether an implementation of a component conforms with the interface specification. Incorrect or nonconforming behavior of the component under a given scheduling is reported as an error by the testing framework.

Due to message delays and overtaking, the order in which outgoing messages from a component are observed by an external observer does not necessarily reflect the order in which they were actually sent. Testing is based on behavior observable at the interface, and the order of outgoing communication should therefore not affect the test results. The operational semantics of the specification language takes the asynchronous nature of the communication model into consideration by treating certain reorderings of output events as observationally equivalent, and testing is done up-to *observational equivalence*.

Reordering of output events can be expressed by defining sequences of output events as *associative* and *commutative*. We argue that our testing framework is especially well suited to implement this since, using the rewriting logic system Maude, associativity and commutativity can be declared using *equational attributes* [9] which allows efficient evaluation of such specifications.

This paper extends [19] which introduced and gave the formal basis for the approach to testing that we explore further here, the main contributions are:

Verification We provide an implementation in the rewriter Maude and use Maude's *search* functionality for state exploration (for rewriting modulo AC) for verification of components and investigate how the support for AC reasoning built in into Maude contributes to state space reduction in verification of asynchronously communicating components.

Experimental results We present *experimental results* from using the Maude rewriting tool which give empirical evidence of the benefits of our method. We compare, in two series of experiments, the influence on the state space of using Maude's built in AC support against explicit representation of all possible reorderings of output events. Using AC rewriting may considerably reduce the resource consumption when testing asynchronously communicating objects. AC rewriting significantly pays off in terms of time and the number of rewrites.

We review the formalisation of Creol in Sect. 2, some technicalities from the previous paper are repeated when necessary. The corresponding behavioral interface specification language and an explanation of how this is used for testing are given in Sect. 3. In Sect. 4, we describe the executable implementation of the theory. The *experimental results* are in Sect. 5.

2 The Creol modeling language

We formalise Creol, a high-level, object-oriented modelling language for distributed systems, Creol features active objects and asynchronous method calls.

In contrast with object-oriented languages based on multi-threading, such as *Java* or $C^{\#}$, the language features *active objects*. The unit of activity is the object; every process belongs to an object, and activity does not cross object borders. Communication is based on exchanging messages *asynchronously*, and is *asymmetric* in the sense that there are linguistic means to *send* a message, but not to *accept* a message: objects are always input-enabled. On the callee side of a method call therefore each object possesses an input "queue" in which incoming messages are waiting to be served by the object. To avoid uncontrolled interference, each object acts as a *monitor*; at most one method body is executing at each point in time. By default the choice of which method call in the input queue that enters the object next is *non-deterministic*. After the abstract syntax, we sketch the operational semantics, concentrating on the external behavior, i.e., the message exchange with the environment.

2.1 Syntax

The abstract syntax, in the style of standard object calculi, is given in Tab. 1. Names n represent references to classes, to objects, and to threads. To facilitate reading, we allow ourselves to write o and its syntactic variants for names referring to objects, c for classes, and n when being unspecific. A *component* C is a collection of classes, objects, and (named) threads, with $\mathbf{0}$ representing the empty component. The sub-entities of a component are composed using the parallel-construct $\|$. The entities executing in parallel are the named threads $n\langle t\rangle$, where t is the code being executed and n the name of the thread. The name n of the thread is at the same

C	$::=$	$\mathbf{0} \mid C \parallel C \mid \nu(n{:}T).C \mid c[\![F, M]\!] \mid o[c, F, L] \mid n\langle t\rangle$	component
F	$::=$	$l = f, \ldots, l = f$	fields
M	$::=$	$l = m, \ldots, l = m$	method suite
m	$::=$	$\varsigma(n{:}T).\lambda(x{:}T, \ldots, x{:}T).t$	method
f	$::=$	$\varsigma(n{:}T).\lambda().v \mid \varsigma(n{:}T).\lambda().\bot_{n'}$	field
t	$::=$	$v \mid \mathsf{stop} \mid \mathsf{let}\, x{:}T = e\, \mathsf{in}\, t$	thread
e	$::=$	$t \mid \mathsf{if}\, v = v\, \mathsf{then}\, e\, \mathsf{else}\, e \mid \mathsf{if}\, \mathit{undef}(v.l())\, \mathsf{then}\, e\, \mathsf{else}\, e$	expr.
	$\mid$	$v@l(\boldsymbol{v}) \mid v.l(\boldsymbol{v}) \mid v.l() \mid v.l := \varsigma(s{:}T).\lambda().v$	
	$\mid$	$\mathsf{new}\, n \mid \mathsf{claim}@(n, n) \mid \mathsf{get}@n \mid \mathsf{suspend}(n) \mid \mathsf{grab}(n) \mid \mathsf{release}(n)$	
v	$::=$	$x \mid n \mid ()$	values
L	$::=$	$\bot \mid \top$	lock status

Table 1
Abstract syntax

time the future reference under which the result value of t, if any, will be available. In this paper we restrict ourselves to the situation where the component consists of one object only, plus arbitrary many threads. A class $c[\![F, M]\!]$ carries a name c and defines its fields and methods in F and M. An object $o[c, F, L]$ with identity o keeps a reference to the class c it instantiates, stores the current value F of its fields, and maintains a *binary lock* L indicating whether any code is currently active inside the object (in which case the lock is taken) or not (in which case the lock is free). The symbols $\top$ and $\bot$ indicate that the lock is taken or free respectively.

The named threads $n\langle t\rangle$ are incarnations of method bodies "in execution". Each thread belongs to one specific object "inside" which it executes, i.e., whose instance variables it has access to. Built in object locks are used to rule out unprotected concurrent access to the object states: Though each object may have more than one method body incarnation partially evaluated, at each time point at most one of those bodies (the lock owner) can be active inside the object. The ν-operator is used for hiding and dynamic scoping, as known from the π-calculus.

Besides components, the grammar specifies the lower level syntactic constructs, in particular, methods, expressions, and (unnamed) threads, which are basically sequences of expressions. The further expressions claim, get, suspend, grab, and release deal with synchronization. They take care of releasing and acquiring the lock of an object appropriately. All of the features and their representation is pretty standard and (apart from the communication via method calls) not visible at the interface, we omit further details here and refer to the technical report [20].

2.2 Operational semantics

The operational semantics of a program being tested is given in two stages: steps *internal* to the program, and those occurring at the interface.

The internal rules deal with steps not interacting with the object's environment, such as sequential composition, conditionals, field look-up and update, etc. The rules are standard and we omit them here. More interesting and relevant are the "external" rules which describe the interaction of a component with its environment, by exchanging communication labels. The communication labels, the basic building blocks of the interface interactions, are given in Tab. 2. A component or object exchanges information with the environment via *call*- and *return*-labels, and the interactions is either incoming or outgoing (marked ? resp. !). The label

$n\langle call\ o.l(\boldsymbol{v})\rangle$ represents a call of method l in object o. In that label, n is a name identifying the thread that executes the method in the callee and is therefore the (future) reference under which the result of the method call will be available (if ever) for the caller. The incoming label $n\langle return(v)\rangle?$ hands the value from the corresponding call back to the object, which renders it ready to be read. Its counterpart, the outgoing return, passes the value to the environment. Besides that, labels can be prefixed by bindings of the form $\nu(n{:}T)$ which express freshness of the transmitted name, i.e., scope extrusion. .

The interface behavior is given by rules as those of Tab. 3 (we show 2 of the four rules, dealing with incoming communication, the missing 2 for outgoing communication are similar). The external steps are given as transitions of the form $\Xi \vdash C \xrightarrow{a} \acute{\Xi} \vdash \acute{C}$, where Ξ and $\acute{\Xi}$ represents the assumption/commitment contexts of C before and after the step, respectively. In particular, the context contains the identities of the objects and threads known so far, and the corresponding typing information. This information is *checked* in incoming communication steps, and updated when performing a step (input or output). These two operations are captured by the following notation

$$\Xi \vdash a : T \qquad \text{and} \qquad \Xi + a \tag{1}$$

which constitute part of the rule premises in Tab. 3. Intuitively, they mean the following: label a is well-formed and well-typed wrt. the information Ξ and refers to an asynchronous call which results in a value of type T. The right-hand notation of (1) extends the binding context Ξ by the bindings transmitted as part of label a appropriately. For lack of space, we omit the formal definitions here. Intuitively, they make sure that only well-typed communication can occur and that the context is kept up-to date during reduction. Rule CallI deals with incoming calls, and basically adds the new thread n (which at the same time represents the future reference for the eventual result) in parallel with the rest of the program. The notation $M.l(o)(\boldsymbol{v})$ represents the parameter passing of the actual values to the

$\gamma ::= n\langle call\ n.l(\boldsymbol{v})\rangle \mid n\langle \mathsf{return}(n)\rangle \mid \nu(n{:}T).\gamma$	basic labels
$a ::= \gamma? \mid \gamma!$	input and output labels

Table 2
Structured communication labels

$$\frac{a = \nu(\Xi').\ n\langle call\ o.l(\boldsymbol{v})\rangle? \quad \Xi \vdash a : T \qquad \acute{\Xi} = \Xi + a}{\Xi \vdash C \parallel o[c, F, \bot] \xrightarrow{a} \acute{\Xi} \vdash C \parallel o[c, F, \top] \parallel n\langle \mathsf{let}\ x{:}T = M.l(o)(\boldsymbol{v})\ \mathsf{in}\ \mathsf{release}(o); x\rangle}\ \textsc{CallI}$$

$$\frac{a = \nu(\Xi').\ n\langle return(v)\rangle? \qquad \Xi \vdash a : ok \qquad \acute{\Xi} = \Xi + a}{\Xi \vdash C \xrightarrow{a} \acute{\Xi} \vdash C \parallel n\langle v\rangle}\ \textsc{RetI}$$

Table 3
External steps

method body t, where s is the "self"-parameter, which is substituted by the identity o of the callee. We write $\Xi_1 \vdash C_1 \stackrel{t}{\Longrightarrow} \Xi_2 \vdash C_2$ if $\Xi_1 \vdash C$ reduces in a number of internal and external steps to $\Xi_2 \vdash C_2$, exhibiting t as the trace of the external steps.

3 A behavioral interface specification language

The behavior of an object in a particular execution is, at the interface, described by a sequence of labels as given by Tab. 2. The black-box behavior of an object can therefore be described by a set of *traces*, each consisting of a finite sequence of labels. This would be the same also for a component consisting of a set of objects, for that matter. To specify sets of label traces, we employ a simple trace language with prefix, choice and recursion. Table 4 contains its syntax. The syntax of the labels in the specification language, naturally, quite resembles the labels of Tab. 2. Comparing Tabs. 2 and 4, there are two differences: first, instead of names or references n, the specification language here uses variables. Second, the labels here allow a binding of the form $(x{:}T).\gamma$, which has no analog in Tab. 2; the form $\nu(x{:}T).\gamma$ corresponds to $\nu(n{:}T).\gamma$, of course. Both binding constructs act as variable declarations, with the difference that $\nu(x{:}T).\gamma$ not just introduces a variable, but in addition asserts that the names represented by that variable must be fresh. The binding $(x{:}T).\gamma$ corresponds to a conventional variable declaration, introducing the variable x which represents arbitrary values.

The grammar given in Tab. 4 allows to specify sets of traces. Not all specifications, however, are meaningful. We rule out ill-formed specifications by introducing restrictions on: *typing:* Values handed over must correspond to the expected types for that methods; *scoping:* Variables must be declared before their use; and *communication patterns:* No value can be returned before a matching outgoing call has been seen at the interface. In addition we take care to consider the *polarity* of the specification. In the specification, it is important to distinguish between input and output interactions, as input messages are under the control of the environment, whereas the outputs are to be provided by the object as specified. This splits the specification into an *assumption* part under the responsibility of the environment, and a commitment part, controlled by the component. To specify non-deterministic behavior, the language supports a choice operator, and we distinguish between choices taken by the environment—external choice—and those the object is responsible for—internal choice. Especially, we do not allow so-called mixed choice. Cf. [20] for details about the formalization of these restrictions, presently just note that it is required that specifications are well-formed, and $\Xi \vdash \varphi : wf^p$ stands for the corresponding judgment. The metavariable p (for polarity) stands for either ?, !, or

$\gamma ::= x\langle call\ x.l(\boldsymbol{x})\rangle \mid x\langle \mathsf{return}(x)\rangle \mid \nu(x{:}T).\gamma \mid (x{:}T).\gamma$	basic labels
$a ::= \gamma? \mid \gamma!$	input and output labels
$\varphi ::= X \mid \epsilon \mid a\ .\ \varphi \mid \varphi + \varphi \mid \mathsf{rec}\ X.\varphi$	specifications

Table 4
Specification language

$$\frac{}{\nu(\Xi)\,.\,\gamma_1!\,.\,\gamma_2!\,.\,\varphi \equiv_{obs} \nu(\Xi)\,.\,\gamma_2!\,.\,\gamma_1!\,.\,\varphi}\ \text{Eq-Switch}$$

$$\frac{\vdash (\varphi_1 + \varphi_2) : wf^!}{\gamma!\,.\,(\varphi_1 + \varphi_2) \equiv_{obs} \gamma!\,.\,\varphi_1 + \gamma!\,.\,\varphi_2}\ \text{Eq-Plus} \qquad \mathsf{rec}\ X.\varphi \equiv_{obs} \varphi[\mathsf{rec}\ X.\varphi/X] \qquad \text{Eq-Rec}$$

Table 5
Observational equivalence

?!, where ?! indicates the polarity for an empty sequence or for a process variable, and ? and ! indicate well-formed input and output specifications respectively.

3.1 Observational blur

Creol objects communicate asynchronously and the order of messages might not be preserved during communication. The order observed by an external observer or tester does not necessarily reflect the order in which the messages were sent, therefore an observed "wrong" order of communication should not be taken to be an error and we must relax the specification up-to some appropriate notion of *observational equivalence*, denoted by $\equiv_{obs}$ and defined by the rules of Tab. 5. Note that the purpose is not to reconstruct some "correct" order of communication. When testing a component, we control the communication, the test specification and framework plays the role of both environment (generating input to the CUT) and observer (controlling output), but want to retain the external perspective in order to test up-to observability. When testing a given object, we specify the order in which the inputs are consumed by the object, rather than the time they have been generated. In this way we specify the input scheduling of the object, which makes our specifications more expressive than in the case of blurring input. At the same time, we specify the outputs of the object as seen from the environment. We therefore blur the output, but not the input. This setting allows synchronous parallel composition. Input blur may be beneficial in other settings, and has e.g. been applied in a reasoning system [14] for Creol based on Hoare logic. In the presented compositional reasoning system, message generation is considered observable, but not messages consumption. Hence, in that system, input is blurred, but not output.

Rule Eq-Switch captures the asynchronous nature of communication, in that the order of outgoing communication does not play a role. Rule Eq-Plus allows to distribute an output over a non-deterministic choice. Rule Eq-Rec expresses the standard unrolling of recursive definitions. The operational semantics of the specification language is straightforward reduction.

3.2 Asynchronous testing of objects

Table 6 defines the interaction of the interface specification, φ, with the component, basically by synchronous parallel composition. Both φ and the component must engage in corresponding steps, which, for incoming communication schedules the order of interactions with the component whereas for outgoing communication the interaction will take place only if it matches an outgoing label in the specification

$$\frac{\Xi \vdash C \xrightarrow{\tau} \Xi \vdash \acute{C}}{\Xi \vdash C \parallel \varphi \to \Xi \vdash \acute{C} \parallel \varphi}\ \textsc{Par-Int} \qquad \frac{\vdash a \lesssim_\sigma b \quad \Xi_1 \vdash C \xrightarrow{a} \acute{\Xi}_1 \vdash \acute{C} \qquad \Xi_1 \vdash \varphi \xrightarrow{b} \acute{\Xi}_2 \vdash \acute{\varphi}}{\Xi_1 \vdash C \parallel \varphi \to \acute{\Xi}_1 \vdash \acute{C} \parallel \acute{\varphi}\sigma}\ \textsc{Par}$$

$$\frac{\Xi \vdash \varphi : wf^?}{\Xi \vdash \nu(\Xi').(C \parallel n\langle \mathsf{let}\ x{:}T = o.l(\boldsymbol{v})\ \mathsf{in}\ t\rangle \parallel \varphi) \to \lightning}\ \textsc{Err-Call} \qquad \frac{\Xi \vdash \varphi : wf^?}{\Xi \vdash \nu(\Xi').(C \parallel n\langle v\rangle \parallel \varphi) \to \lightning}\ \textsc{Err-Ret}$$

Table 6
Parallel composition

and an error is raised if input is required by the specification. The component can proceed on its own via internal steps (cf. rule Par-Int). Rule Par requires that, in order to proceed, the component and the specification must engage in the "same" step, where φ's step b is matched against the step a of the component. Here $\vdash a \lesssim_\sigma b$ states that there exist a substitution σ such that the label a produced by the component and the label b specified by the interface description can be matched. Note that after a successful application of the Par rule, variables in the specification may have been substituted with concrete values. We omit the details of the matching and refer to the technical report [20]. The rules Err-Call and Err-Ret report an error if the specification requires an input as the next step and the object however could do an output, either a call or a return. In the rule $\lightning$ indicates the occurrence of an error. Note that the equivalence relation, according to the rule Eq-Switch, allows the reordering of outputs, but not of inputs.

4 A specification-driven interpreter for Creol

The operational semantics of Creol is formalized in rewriting logic [31] and executable on the Maude rewriting engine [8], this gives an interpreter for Creol. Our executable framework for testing Creol components includes: the specification language formalized in rewriting logic and a modified version of the Creol interpreter. We obtain a *specification-driven interpreter* for testing by synchronizing the communication between specification terms and objects. Input to the component is generated non-deterministically within the bounds of the specification, and at the same time it is tested that the output behavior of the object conforms to the specification, the internal activity is unmodified compared to the standard interpreter. The default behavior for Creol is to place incoming method calls into the callee's input queue from which calls are non-deterministically selected for execution. For the specification-driven interpreter if an incoming call is specified and the lock of the object is free the corresponding method code should start executing immediately. In the implementation the incoming messages are generated directly from the specification.

Standard simulation of a Creol model in Maude is achieved by rewriting an intial model configuration together with the interpreter. Maude's *search* command may also be used to search for specific result configurations. For *testing* a component, instead of using the initial configuration as input, we extract from the model one object and its class definitions. This becomes the component under test (CUT).

The CUT, its specification and the modified interpreter is then rewritten by Maude. Thus specific behavioral properties of selected objects from a large model may be tested. A standard Creol state configuration (Cfg) is a multiset of objects, classes, and messages and the Maude rewrite rules for transitions are of the form `rl Cfg => Cfg'`. For the specification-driven interpreter, we introduce terms `Spec` for specifications and add rules on the form `(Spec || O) Cfg => (Spec' || O') Cfg'` to test the object `O` with respect to `Spec`, where `||` represents the synchronous parallel composition. Each rule evolves the state of a specification and the state of an object in a synchronized manner: an interaction only takes place when it matches a complementary label in the specification. E.g., the Par rule in Tab. 6 is implemented by several Maude rules for the different kinds of communication events that may occur. We refer the reader to [19] for some examples.

In the implementation, we define associative and commutative (AC) output prefixes by declaring the prefix operator to be AC in the cases where an output label is prefixed to an output specification. Together with a Maude rule that implements distribution over choice (the rule Eq-Plus above), this enables the testing framework to do testing up-to observational equivalence.

5 Experimental results

This section describes two series of experiments, using the implementation sketched in the previous section. The experiments demonstrate the usefulness of the approach: using AC rewriting may considerably reduce the resource consumption, when testing asynchronously communicating objects. AC rewriting significantly pays off in terms of time and the number of rewrites. With regards to the state space, the effects are not so definite.

The first example is tailor-made to show the effects for a simple component. The second example is an abstracted version of the "loan quote example" known from the area of enterprise application integration [23]. The examples also illustrate how to use the interface specification language for testing component behavior and how to employ model checking via the *search* command of Maude to also achieve *verification* of a component with a trace specification. When using the search command, Maude not just explores *one* trace, but explores the set of behaviors given by the component together with the interface trace description. That the system in general explores a set of traces, as opposed to just one, has the following reasons: first, exploring a trace (trivially) means exploring all prefixes; that, of course, does not only apply to using Maude's search, but to simple rewriting as well. Second, the specification may contain non-determinism (besides the fact that also the component may behave non-deterministically). Finally, and most important in our context, one trace is always meant up-to the "observational blur", as specified in Tab. 5.

To measure the effect of AC rewriting, both series of experiments are carried out two times, either with AC rewriting switched on, or else off. When AC equivalence on the specification is switched off, we use an equivalent but expanded version of the specification to compare the results.

In the first example, the component under test consists of one object with n methods m_1 through m_n. The specification prescribes that all methods must have

been called before any method may return. In Creol this is implemented by combining processor release points and *await* guards [27]. The behavioral specification for 3 methods reads:

$$\varphi_{c3} = n_1\langle call\ c.m_1(x_1)\rangle?\ .\ n_2\langle call\ c.m_2(x_2)\rangle?\ .\ n_3\langle call\ c.m_3(x_3)\rangle?\ .\\ (n_1\langle return(y_1)\rangle!\ .\ n_2\langle return(y_2)\rangle!\ .\ n_3\langle return(y_3)\rangle!)\ .\ \epsilon$$

A test is executed by giving the Maude command: `rew` (φ_{c3} `|| c) cClass .`, where c represents the Creol object. Maude rewrites the configuration, either resulting in an error reported when the component is about to execute an unspecified output, or stopping when no more rules apply. In the latter case, if the original specification is fully consumed this gives evidence that the component conforms to the specification, in the sense that test execution of c only leads to output foreseen by the specification φ_{c3}. This conformance relation is similar to the input-output conformance relation (**ioco**) of [35].

Definition 5.1 Let $out(\varphi\ after\ t)$ represent the set of all possible output events that is specified by φ after execution of the trace t. Let $out(c\ after\ t)$ represent the set of possible output events for the component c after execution of t. Let $traces(\varphi)$ be the set of traces that the specification designates. Our conformance relation *conf* is defined as follows:

$$c\ conf\ \varphi \Leftrightarrow_{def} \forall t \in traces(\varphi) :\ out(c\ after\ t) \subseteq out(\varphi\ after\ t)$$

Depending on the internal interleaving of the threads initiated by the method calls, different outcomes are possible. Maude's *search* command can be used to do a breadth first search for error configurations in the reachable state space:

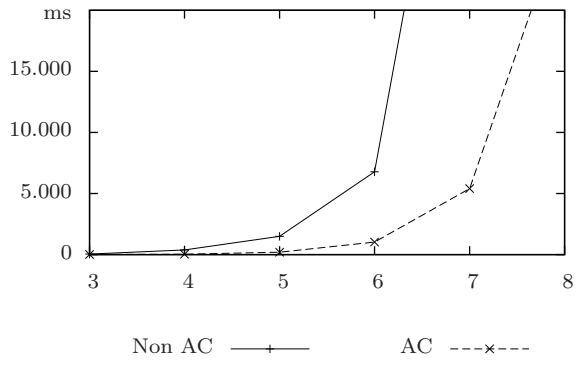
```
search  in PROGRAM :  φc3 || c cClass =>+
                        φ || conf errorMsg(S:String) .
```

By altering the order of the input labels in the specification, we can easily check how different scheduling of input affect the execution of the object. E.g., a search for error states from a specification φ'_{c3} where the order of calls are to m_1, m_3, and m_2 gives no solutions, which means that with the methods called in this order, the component cannot fail to conform to the specification.

The two series of data, plotted in Fig. 1, show the time needed for exploring the state space with or without AC rewriting, where n is the number of methods. The figures show that with AC rewriting the increase in number of rewrites is considerably less than using the equivalent, expanded version of the specification.

In the second example, a *broker* acts as an intermediary between a client and several providers of some service (cf. [23]). Initially we consider a broker that after being requested to do so by a client queries a fixed number of providers for a (price) quote and returns an answer to the client giving the best alternative found. A specification for a broker querying two service providers can be given as:

$$\varphi_b = n_{c1}\langle call\ b.getP(x)\rangle?\ .\\ (\ n_1\langle call\ p_1.getQ(x)\rangle!\ .\ n_2\langle call\ p_2.getQ(x)\rangle!\ )\ .\\ n_1\langle return(v_1)\rangle?\ .\ n_2\langle return(v_2)\rangle?\ .\ n_{c1}\langle return(v)\rangle!\ .\ \epsilon.$$



n	ms CPU time	
	AC	*Non AC*
3	16	47
4	38	379
5	198	1.498
6	1.030	6.782
7	5.407	49.311
8	27.894	NA
9	153.316	NA

Fig. 1. Validation of c with and without AC rewriting.

Note that whereas the previous example illustrated generation of incoming calls to the component and testing of outgoing returns from the component, this example also includes testing of outgoing calls, and generation of incoming returns. For incoming returns, the test framework generates pseudo-random, type correct return values. For this specification a broker component would be non-conforming if it were to call the providers before receiving a call from the client and also if it were to return the initial call from the client before finishing its interaction with the providers.

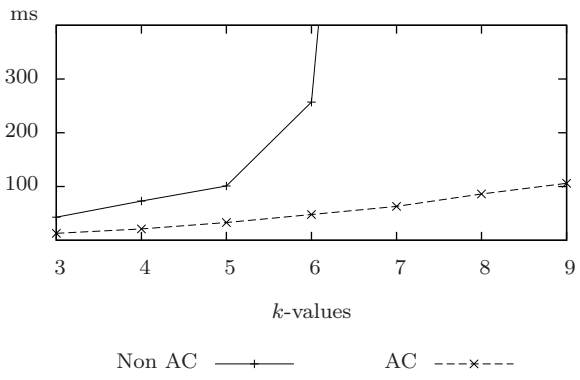
In an open setting, the number of providers that a broker knows is likely to change over time, hence we assume that a broker will be notified by new providers and establish connections with them as well as losing connections with others. A further developed version of the broker supports this by allowing the client to give the number of providers that the broker must query before giving a response as a parameter to the call to the method *getP*. The method *getP* now takes two parameters, the name of the service for which a quote is requested, and the number of providers the broker should contact. To validate the behaviour of this new broker we use a series of specifications on the following form

$$\begin{aligned}\varphi_{bk} = {} & n_{c1}\langle \mathit{call}\ b.\mathit{getP}(x,k)\rangle?\ .\\ & (\mathit{provider\ registration})\ .\\ & (\ n_1\langle \mathit{call}\ p_1.\mathit{getQ}(x)\rangle!\ .\ \ldots\ .\ n_k\langle \mathit{call}\ p_k.\mathit{getQ}(x)\rangle!\ )\ .\\ & n_1\langle \mathit{return}(v_1)\rangle?\ .\ \ldots\ .\ n_k\langle \mathit{return}(v_k)\rangle?\ .\ n_{c1}\langle \mathit{return}(v)\rangle!\ .\ \epsilon\ ,\end{aligned}$$

where k is the number of providers. Figure 2 plots the times of AC rewriting, resp. explicit rewriting against k.

6 Conclusion

We have presented a formalization of a concurrent object-oriented language and a behavioral specification language, for testing and validation of asynchronously communicating objects. Potential reorderings of communication events occur due to network properties. Our approach describes one way to deal with such situations, namely by defining rewriting specifications modulo AC for output events. One advantage of this approach is that we can define precisely the scheduling of



k	ms CPU time	
	AC	*Non AC*
3	13	43
4	21	73
5	33	101
6	48	257
7	63	1.965
8	86	17.796
9	106	NA

Fig. 2. Validation of the broker component

input, and test internal synchronization properties of the object. When evaluating our approach by experimental case studies we get evidence that using modulo AC rewriting enable us to cover more extensive test cases than we could do otherwise.

Testing of Creol models is relevant also for testing of implementations in languages like C or Java: First indirectly, since many forms of non-determinism inherent in distributed system can be formalized by means of associativity and commutativity, our results are relevant also for other languages with asynchronous communication, and for alternative definitions of observational equivalence. Second, and more directly, in [22] and [1] it is shown how different testing techniques can be employed to check for conformance between a Creol model and an industrial distributed system implemented in C. In [22] the technique of dynamic symbolic execution is used to test for conformance between the Creol model and the implementation. Using the same case study, the authors of [1] show how to instrument existing Creol models for testing. Aspect-C is used to insert event recording points into the existing code of the SuT. The model is likewise instrumented with synchronisation points. A tester process is used to replay the recorded events in the model and synchronises with events recorded by the tester, only allowing the model to proceed beyond synchronisation points if the corresponding event was recorded in the SuT. Thus conformance of implementation and Creol model may be verified. Combining these methods with our method for verification of conformance between the Creol model and the specification yields a method for conformance testing of implementations against a specification.

6.1 Related work

Systematic testing is indispensable to assure quality of software and systems. [6] presents an approach to integrate black-box and white-box testing for object-oriented programs. Equivalence is based on the idea of observably equivalent terms and fundamental pairs as test cases, but not in an asynchronous setting.

Godefroid et.al. [18] describe how state-space reductions can be achieved for *input* sequences in the context of constraint-based programming languages. A test algorithm is proposed which systematically generates all possible behavior by selecting input events non-deterministically from a predefined set. By exploiting the inability of constraint languages to observationally distinguish permutations of un-

ordered sets of inputs, the combinatorial explosion is reduced, and a significantly more effective test algorithm is presented. A main difference from our approach is that the reduction in the state space is derived from the structure of the constraint-program itself and not from commutativity of the communicated events. The testing process is driven by the state-space exploration tool VeriSoft [17].

The paper [13] describes compositional analysis based on combining components with specifications. Also here VeriSoft is used for bounded model checking of assume/guarantee specifications, built-in partial order reduction contributes to efficiency of the analysis. However, both the object interaction model, shared variables, and the specifications, invariant based, using Hoare logic, differ from ours.

In [3] assumptions are used as environments to drive individual components for unit testing. LTSs are used to model the behavior of components. An interesting feature of this work, absent in ours, is techniques for automatic generation of exactly the assumptions that a component needs to make about the environment for some property to hold.

Testing for *concurrent* object-oriented programs based on synchronization sequences is investigated in [7], using Petri nets and OBJ as foundation. In his thesis [29], Long presents ConAn ("concurrency analyser"), which generates test drivers from test scripts. The method allows to specify sequences of component method calls and the order in which the calls should be issued (see also [30,33]). For scheduling the intended order, an external *clock* is used, introduced for the purpose of testing. The NModel-framework, comprehensively covered in [25], offers model-based analysis and model-based testing for $C^{\#}$, where abstract models, generally speaking transition systems, of object-oriented programs are used for testing. Related and likewise developed at Microsoft is the Spec Explorer approach (and its predecessor AsmLT), a tool for testing reactive, object-oriented programs. Underlying the model programs, given e.g., in the *Spec*$^{\#}$ specification language, are "model automata" which can be seen as a combination of interface automata and abstract state machines (ASMs), and which are used for test case generation. Dealing with non-determinism, the models separate observable and controllable actions, similar as we distinguish between inputs and output actions in our specification language. Relying on game theoretic foundations, their notion of conformance is based on alternating simulation, not on comparing traces, as in this work. To cope with large and potentially infinite state spaces, Spec Explorer uses different abstraction and pruning techniques. One is based on building a quotient of the model automaton by identifying states which are considered equivalent ("state groupings", cf. [21] and [5]). These state groupings correspond to predicate abstraction known from model checking and serve a similar purpose as the observable equivalence presented here. I.e., the are used to reduce the state space, but are user-given and not specifically capturing observably equivalent states due to asynchronous communication. For a thorough discussion of Spec Explorer and links to further results in that context, see [37].

Another well-established approach for functional testing is input/output conformance testing (ioco for short) [34,35]. Ioco is based on input-output transition systems, our conformance relation is closely related. Component-based testing and testing in context, using the ioco test theory, are studied in [36]. A number of

test-tools are based on variants of the ioco test theory, such as TGV, TestGen, and TorX. In the context of ioco testing, [15] uses *symbolic* transition systems to counter the state explosion problem. Unit testing framework for actors, i.e., active concurrent objects, is presented in [10], using the discrete event based simulation environment OPNET. Validation of component interfaces specified in rewriting logic is the subject also of [26]. [32] considers Creol and investigates how different scheduling of object activity restrict the behavior. The focus is on *intra*-object scheduling, and on test purposes as assertions on the *internal* state of the object. This is in contrast to our focus on the interface communication.

6.2 Future work

Creol has successfully been used to model complex and highly dynamic communication systems, e.g. wireless sensor networks in [28], where the Ad hoc On-Demand Distance Vector (AODV) routing algorithm is used as a case study. ASK is an industrial size multi-threaded, asynchronous application for connecting people. A substantial part of ASK has been modelled in Creol [1]. Both these models are complex. The similarity of Creol and an object-oriented programming language, and Creol's expressiveness allow for models that are structurally close to the AODV algorithm resp. the ASK system itself. This leads to a need for testing the models. We are currently working on applying our method for model-based testing of Creol models to the AODV model.

Acknowledgement
We thank Rudolf Schlatte for insight into application testing with Creol, and the anonymous referees for constructive criticism and hints to related work.

References

[1] Bernhard Aichernig, Andreas Griesmayer, Rudolf Schlatte, and Andries Stam. Modeling and testing multi-threaded asynchronous systems with Creol. In *TTSS'08*, volume 243 of *ENTCS*. Elsevier, 2009.

[2] G. Bernot. Testing against formal specification: A theoretical view. In Samson Abramsky and T. S. E. Maibaum, editors, *TAPSOFT '91, Volume 1*, volume 493 of *LNCS*, pages 99–119. Springer, 1991.

[3] C. Blundell, D. Giannakopoulou, and C. S. Pasareanu. Assume-guarantee testing. In *Proceedings of SAVCBS'05*, pages 7–14, 2005.

[4] M. Broy, B. Jonsson, J.-P. Katoen, M. Leucker, and A. Pretschner, editors. *Model-Based Testing of Reactive Systems*, volume 3472 of *LNCS*. Springer, 2005.

[5] Colin Campbell and Margus Veanes. State exploration with multiple state groupings. In *ASM'05*. Laboratory of Algorithms, Complexity, and Logic, University Paris 12, 2005.

[6] H. Y. Chen, T. H. Tse, F. T. Chan, and T. Y. Chen. In black and white: An integrated approach to class-level testing of object-oriented program. *ACM Transactions of Software Engineering and Methodology*, 7(3):250–295, 1998.

[7] Huo Yan Chen, Yu Xia Sun, and T. H. Tse. A strategy for selecting synchronization sequences to test concurrent object-oriented software. In *COMPSAC '03*. IEEE Computer Science Press, 2003.

[8] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn Talcott. The Maude 2.0 system. In *RTA 2003*, volume 2706 of *LNCS*. Springer, June 2003.

[9] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn Talcott. *The Maude Manual (version 2.1.1)*. SRI International, Menlo Park, April 2005.

[10] Mark E. Coyne, Scott R. Graham, Kenneth M. Hopkinson, and Stuart H. Kurkowski. A methodology for unit testing actors in proprietary discrete event based simulations. In *WSC '08*. Winter Simulation Conference, 2008.

[11] The Creol language. http://heim.ifi.uio.no/creol.

[12] S. R. Dalal, A. Jain, N. Karunanithi, J. M. Leaton, C. M. Lott, G. C. Patton, and B. M. Horowitz. Model-based testing in practice. In *Proc. of the 1999 Intl. Conference on Software Engineering*, 1999.

[13] Juergen Dingel. Computer-assisted assume/guarantee reasoning with verisoft. In *25th International Conference on Software Engineering (ICSE'03)*, 2003.

[14] Johan Dovland, Einar Broch Johnsen, and Olaf Owe. Observable behavior of dynamic systems: Component reasoning for concurrent objects. In *FInCo '07*, volume 203 of *ENTCS*. Elsevier, 2008.

[15] Lars Frantzen, Jan Tretmans, and Tim A. C. Willemse. Test generation based on symbolic specifications. In J. Grabowski and B. Nielsen, editors, *FATES 2004*, pages 1–15, 2005.

[16] Marie-Claude Gaudel. Testing can be formal, too. In Peter D. Mosses, Mogens Nielsen, and Michael I. Schwarzbach, editors, *Proceedings of TAPSOFT '95*, volume 915 of *LNCS*, pages 82–96. Springer, 1995.

[17] Patrice Godefroid. Model checking for programming languages using VeriSoft. In *Proceedings of POPL '97*, pages 174–186. ACM, January 1997.

[18] Patrice Godefroid, L. Jagadeesan, R. Jagadeesan, and K. Läufer. Automated systematic testing for constraint-based interactive services. In *SIGSOFT FSE*. ACM, 2000.

[19] Immo Grabe, Marcel Kyas, Martin Steffen, and Arild B. Torjusen. Executable interface specifications for testing asynchronous Creol components. In *FSEN '09*, volume 5961 of *LNCS*. Springer, 2010.

[20] Immo Grabe, Marcel Kyas, Martin Steffen, and Arild Braathen Torjusen. Executable interface specifications for testing asynchronous Creol components. Tech. Report 375, Univ. of Oslo, July 2008.

[21] W. Grieskamp, Y. Gurevitch, W. Schulte, and M. Veanes. Generating finite state machines for abstract state machines. In *ISSTA'07, vol. 27 of Software Engineering Notes*. ACM, 2002.

[22] Andreas Griesmayer, Bernhard Aichernig, Einar Broch Johnsen, and Rudolf Schlatte. Dynamic symbolic execution of distributed concurrent objects. In *FMOODS/FORTE'09*, volume 5522 of *LNCS*. Springer, June 2009.

[23] Gregor Hohpe and Bobby Woolf. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley, 2003.

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

[25] Jonathan Jacky, Margus Veanes, Colin Campbell, and Wolfram Schulte. *Model-Based Software Testing and Analysis with C♯*. Cambridge University Press, 2008.

[26] Einar Broch Johnsen, Olaf Owe, and Arild B. Torjusen. Validating behavioral component interfaces in rewriting logic. *Fundamenta Informaticae*, 82(4):341–359, 2008.

[27] Einar Broch Johnsen, Olaf Owe, and Ingrid Chieh Yu. Creol: A type-safe object-oriented model for distributed concurrent systems. *Theoretical Computer Science*, 365(1–2):23–66, November 2006.

[28] Wolfgang Leister and Joakim Bjørk. Modelling routing algorithms for wireless sensor networks in Creol, 2009. Presented at the 21st Nordic Workshop on Programming Theory, NWPT '09, Copenhagen.

[29] Bradley Long. *Testing Concurrent Java Components*. PhD thesis, University of Queensland, July 2005.

[30] Bradley Long, D. Hofmann, and P. Strooper. Tool support for testing concurrent Java components. *IEEE Transactions on Software Engineering*, 29(6):555–566, 2003.

[31] José Meseguer. Conditional rewriting as a unified model of concurrency. *TCS*, 96:73–155, 1992.

[32] Rudolf Schlatte, Bernhard Aichernig, Frank de Boer, Andreas Griesmayer, and Einar Broch Johnsen. Testing concurrent objects with application-specific schedulers. In *ICTAC '08*, volume 5160 of *LNCS*. Springer, 2008.

[33] Paul Strooper and Luke Wildman. Testing concurrent Java components. In *ICSE COMPANION '07*, pages 161–162, Washington, DC, USA, 2007. IEEE Computer Society.

[34] Jan Tretmans. Test generation with inputs, outputs, and quiescence. In T. Margaria and B. Steffen, editors, *TACAS '96*, volume 1055 of *LNCS*. Springer, 1996.

[35] Jan Tretmans. Test generation with inputs, outputs, and repetetive quiescence. *Software — Concepts and Tools*, 17(3):103–120, 1996.

[36] Machiel van Bijl, Arend Rensink, and Jan Tretmans. Compositional testing with ioco. In *TestCom/FATES 2003*, volume 2931 of *LNCS*, pages 86–100. Springer, 2003.

[37] Margus Veanes, Colin Campbell, Wolfgang Grieskamp, Wolfram Schulte, Nikolai Tillmann, and Lev Nachmanson. Model-based testing of object-oriented reactive systems with spec explorer. In *Formal Methods and Testing*, volume 4949 of *LNCS*. Springer, 2008.