Executable interface specifications for testing asynchronous Creol components*

Immo Grabe[1], Marcel Kyas[2], Martin Steffen[2], and Arild B. Torjusen[2]

[1] Christian-Albrechts University Kiel, Germany
[2] University of Oslo, Norway

Abstract. We propose and explore a formal approach for black-box testing asynchronously communicating components in open environments. Asynchronicity poses a challenge for validating and testing components. We use Creol, a high-level, object-oriented language for distributed systems and present an interface specification language to specify components in terms of traces of observable behavior.
The language enables a concise description of a component's behavior, it is executable in rewriting logic and we use it to give test specifications for Creol components. In a specification, a clean separation between interaction under control of the component or coming from the environment is central, which leads to an assumption-commitment style description of a component's behavior. The assumptions schedule the inputs, whereas the outputs as commitments are tested for conformance with the specification. The asynchronous nature of communication in Creol is respected by testing only up-to a notion of observability. The existing Creol interpreter is combined with our implementation of the specification language to obtain a specification-driven interpreter for testing.

1 Introduction

To reason about open distributed systems and predicting their behavior is intrinsically difficult. A reason for that is the inherent asynchronicity and the resulting non-determinism. It is generally accepted that the only way to approach complex systems is to "divide-and-conquer", i.e., consider components interacting with their environment. Abstracting from internal executions, their black-box behavior is given by interactions at their *interface.* In this paper we use Creol [20], a programming and modeling language for distributed systems based on concurrent, active objects communicating via asynchronous method calls.

To describe and test Creol components, we introduce a concise specification language over communication labels. The expected behavior 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, but

* Part of this work has been supported by the EU-project IST-33826 *Credo: Modeling and analysis of evolutionary structures for distributed services* and the German-Norwegian DAAD-NWO exchange project *Avabi* (Automated validation for behavioral interfaces of asynchronous active objects).

of the environment, input is considered as assumptions about the environment whereas output describes commitments of the object. 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. An expression in the specification language thus gives an assumption-commitment style specification[10] for a component by defining the valid observable output behavior under the assumption of a certain scheduling of the input. Scheduling and testing of a component is done by synchronizing the execution of the component 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 labels 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.

It is important in the specification, to carefully distinguish between the interactions which are scheduled and those for which the component is responsible and which are checked for conformance. We do so by formalizing *well-formedness* conditions on specifications. Well-formedness enforces a syntactic distinction between input and output specifications and, in addition, assures that only "meaningful" traces, i.e., those corresponding to possible behavior, can be specified. Besides that, the specification language captures two crucial features of the interface behavior of Creol objects. First, Creol allows to dynamically create objects and threads (via asynchronous method calls), which gives rise to dynamic scoping. This is reflected in the interface behavior by scope extrusion and the specification language allows to express *freshness* of communicated object and thread references. Second, due to the asynchronous nature of the communication model, the order in which outgoing messages from a component are observed by an external observer does not necessarily reflect the order in which they where actually sent. We take this asynchronous message passing into account by only considering trace specifications up-to an appropriate notion of *observational equivalence*.

Contributions The paper contains the following contributions: We *formalize* the interface behavior of a concurrent, object-oriented, language plus a corresponding behavioral interface specification language in Sect. 2 and Sect. 3. This gives the basis for testing active Creol objects, where a test environment can be simulated by execution of the specifications. Sect. 4 explains how to compose a Creol program and a specification and how to use this for testing. Furthermore, the existing Creol interpreter is extended with the *implementation* of the specification language. This yields a specification-driven interpreter for testing asynchronous Creol components. The implementation is described in Sect. 5

2 The Creol language

Creol [9,20] is a high-level object-oriented language for distributed systems, featuring active objects and asynchronous method calls. Concentrating on the core features, we elide inheritance, dynamic class upgrades, etc. They would complicate the interface description, but not alter the basic ideas presented here.

The Creol-language features active objects and its communication model is based on exchanging messages *asynchronously.* This is in contrast with object-oriented languages based on multi-threading, such as *Java* or $C^{\#}$, which use "synchronous" message passing in which the calling thread inside one object blocks and control is transferred to the callee. Exchanging messages asynchronously decouples caller and callee, which makes that mode of communication advantageous in a distributed setting. On the receiver side, i.e., at the side of the callee, 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*, i.e., at most one method body is executing at each point in time. The choice, which method call in the input queue is allowed to enter the object next is *non-deterministic*.

After presenting the abstract syntax in the next section, we sketch the operational semantics, concentrating on the external behavior, i.e., the message exchange with the environment.

2.1 Syntax

The abstract syntax of the calculus, which is in the style of standard object calculi [1], is given in Tab. 1. It distinguishes between *user* syntax and *run-time* syntax, the latter underlined. The user syntax contains the phrases in which programs are written; the run-time syntax contains syntactic material additionally needed to express the behavior of the executing program in the operational semantics.

The basic syntactic category of names n, which count among the values v, represents 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. Technically, the disambiguation between the different roles of the names is done by the type system and the abstract syntax of Tab. 1 uses the non-specific n for names. The unit value is represented by () and x stands for variables, i.e., local variables and formal parameters, but not instance variables.

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 time, the future reference under which the result value of t, if any, will be available. In this paper, when describing the interface behavior, we restrict ourselves to the situation where the component consists of one object only, plus arbitrary many threads/method

C	$::= \mathbf{0} \mid C \parallel C \mid \underline{\nu(n{:}T).C} \mid n[\![O]\!] \mid \underline{n[n, F, L]} \mid \underline{n\langle t\rangle}$	component
O	$::= F, M$	object
M	$::= l = m, \ldots, l = m$	method suite
F	$::= l = f, \ldots, l = f$	fields
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.
	$\mid\ v@l(\vec{v}) \mid v.l() \mid v.l := \varsigma(s{:}n).\lambda().v$	
	$\mid\ \mathsf{new}\, n \mid \mathsf{claim@}(n, n) \mid \underline{\mathsf{get@}n} \mid \mathsf{suspend}(n) \mid \underline{\mathsf{grab}(n)} \mid \underline{\mathsf{release}(n)}$	
v	$::= x \mid n \mid ()$	values
L	$::= \bot \mid \top$	lock status

Table 1. Abstract syntax

bodies under execution. A class $c[\![O]\!]$ carries a name c and defines its methods and fields in O. 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. Of the three kinds of entities at the component level—threads $n\langle t\rangle$, classes $n[\![O]\!]$, and objects $o[c, F, L]$—only the threads are *active*, executing entities, being the target of the reduction rules. The objects, in contrast, store the state in their fields or instance variables, whereas the classes are constant entities specifying the methods.

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. 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. Method calls in Creol are issued *asynchronously,* i.e., the calling thread continues executing and the code of the method being called is computed concurrently in a new thread located in the callee object. The ν-operator is used for hiding and dynamic scoping, as known from the π-calculus [23]. In a component $C = \nu(n{:}T).C'$, the scope of the name n (of type T) is restricted to C' and unknown outside C. ν-binders are introduced when dynamically creating new named entities, i.e., when instantiating new objects or new threads. The scope of a ν-binder is dynamic, when the name is communicated by message passing, the scope is enlarged.

Besides components, the grammar specifies the lower level syntactic constructs, in particular, methods, expressions, and (unnamed) threads, which are basically sequences of expressions. A method $\varsigma(s{:}T).\lambda(\vec{x}{:}\vec{T}).t$ provides the method body t abstracted over the ς-bound "self" parameter, here s, and the formal parameters $\vec{x}$. For uniformity, fields are represented as methods without parameters

$o[c, F, L] \parallel n\langle \mathsf{let}\, x{:}T = o.l() \,\mathsf{in}\, t\rangle \xrightarrow{\tau} o[c, F, L] \parallel n\langle \mathsf{let}\, x{:}T = F.l(o)() \,\mathsf{in}\, t\rangle$ FLOOKUP

$o[c, F, L] \parallel n\langle \mathsf{let}\, x{:}T = o.l := v \,\mathsf{in}\, t\rangle \xrightarrow{\tau} o[c, F.l := v, L] \parallel n\langle \mathsf{let}\, x{:}T = o \,\mathsf{in}\, t\rangle$ FUPDATE

$n\langle \mathsf{let}\, x : T = o@l(\vec{v}) \,\mathsf{in}\, t\rangle \xrightarrow{\tau}$
$\nu(n'{:}T)(n\langle \mathsf{let}\, x : T = n' \,\mathsf{in}\, t\rangle \parallel n'\langle \mathsf{let}\, x : T = o.l(\vec{v}) \,\mathsf{in}\, \mathsf{stop}\rangle)$ CALLO$_i$

Table 2. Internal steps

(except self), with a body being either a value or yet undefined. Note that the methods are stored in the classes but the fields are kept in the objects. In freshly created objects, the lock is free, and all fields carry the undefined reference $\bot_c$, where class name c is the (return) type of the field.

We use f for instance variables or fields and $l = \varsigma(s{:}T).\lambda().v$, resp. $l = \varsigma(s{:}T).\lambda().\bot_c$ for field variable definition (l is the label of the field). Field access is written as $v.l()$ and field update as $v'.l := \varsigma(s{:}T).\lambda().v$. By convention, we abbreviate the latter constructs by $l = v$, $l = \bot_c$, $v.l$, and $v'.l := v$. Note that the construct $v.l()$ is used for field access only, but not for method invocation. We also use $v_\bot$ to denote either a value v or a symbol $\bot_c$ for being undefined. Direct access to fields across object boundaries is forbidden by convention, and we do not allow method update. Instantiation of a new object from class c is denoted by $\mathsf{new}\, c$.

The expression $o@l(\vec{v})$ denotes an asynchronous method call, where the caller creates a new thread/future reference and continues its execution. 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. As they work pretty standard and as lock-handling is not visible at the interface (and thus does not influence the development), we omit describing them in detail here and refer to the longer version [15].

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 two stages correspond to the rules of Tab. 2 and 4. The internal rules deal with steps not interacting with the object's environment, such as sequential composition, conditionals, field lookup and update, etc. The rules are standard and most are omitted here. We also omit the definition of structural congruence here, specifying standard structural properties such as associativity, commutativity, and basic facts about scoping. The elided rules can be found in the long version [15]. The communication labels, the basic building blocks of the interface interactions, are given in Tab. 3. 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 basic label $n\langle call\ o.l(\vec{v})\rangle$ 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. As usual, the order of such bindings does not play a role

Given a basic label $\gamma = \nu(\Xi).\gamma'$ where Ξ is a name context such that $\nu(\Xi)$ abbreviates a sequence of single $n{:}T$ bindings and where γ' does not contain any binders, we call γ' the *core* of the label and refer to it by $\lfloor\gamma\rfloor$. We define the core analogously for receive and send labels. The free names $fn(a)$ and the bound names $bn(a)$ of a label a are defined as usual, whereas $names(a)$ refer to all names of a.

The interface behavior is given by the 4 rules of Tab. 4, which correspond to the 4 different kinds of labels. 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 rules' premises in Tab. 4. 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. If not interested in the type, we write $\Xi \vdash a : ok$, instead. 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. In the configuration after the reduction step, the meta-mathematical notation $M.l(o)(\vec{v})$ stands for $t[o/s][\vec{v}/\vec{x}]$, when the method suite $[M]$ equals $[\ldots, l = \varsigma(s{:}T).\lambda(\vec{x}{:}\vec{T}).t, \ldots]$. Note that the step is only possible, if the lock of the object is free ($\bot$); after the step, the lock is taken ($\top$). An outgoing call (cf. CallO) is issued by executing

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

Table 3. Communication labels

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

$$\frac{a = \nu(\Xi').\ n\langle \mathit{call}\ o.l(\vec{v})\rangle!\quad \Xi' = \mathit{fn}(\lfloor a \rfloor) \cap \Xi_1 \quad \acute{\Xi}_1 = \Xi_1 \setminus \Xi' \quad \Delta \vdash o \quad \acute{\Xi} = \Xi + a}{\Xi \vdash \nu(\Xi_1).(C \parallel n\langle \mathsf{let}\ x{:}T = o.l(\vec{v})\ \ \mathsf{in}\ \ t\rangle) \xrightarrow{a} \acute{\Xi} \vdash \nu(\acute{\Xi}_1).(C)}\ \textsc{CallO}$$

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

$$\frac{a = \nu(\Xi').\ n\langle \mathit{return}(v)\rangle!\qquad \Xi' = \mathit{fn}(\lfloor a \rfloor) \cap \Xi_1 \quad \acute{\Xi}_1 = \Xi_1 \setminus \Xi' \quad \acute{\Xi} = \Xi + a}{\Xi \vdash \nu(\Xi_1).(C \parallel n\langle v\rangle) \xrightarrow{a} \acute{\Xi} \vdash \nu(\acute{\Xi}_1).C}\ \textsc{RetO}$$

Table 4. External steps

$o.l(\vec{v})$. Furthermore, the binding context Ξ is updated and, additionally, previously private names mentioned in Ξ_1 might escape by scope extrusion, which is calculated by the second and third premise. Remember, that an asynchronous call, as given in $\textsc{CallO}_i$ from Tab. 2 does not immediately lead to an interface interaction, but is an internal step, which only afterwards (asynchronously) leads to the interface interaction as specified in CallO here. Rules RetI and RetO deal with returning the value at the end of a method call.

We write $\Xi_1 \vdash C_1 \overset{t}{\Longrightarrow} \Xi_2 \vdash C_2$ 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 Behavioral interface specification language

The behavior of an object (or a component consisting of a set of objects, for that matter) at the interface is described by a sequence of labels as given by Tab. 3. The black-box behavior of a component can therefore be described by a set of *traces*, each consisting of a finite sequence of labels. To specify sets of label traces, we employ a simple trace language with prefix, choice and recursion. Table 5 contains its syntax. The syntax of the labels in the specification language, naturally, quite resembles the labels of Tab. 3. Comparing Tabs. 3 and 5, there are

$\gamma ::= x\langle \mathit{call}\ x.l(\vec{x})\rangle \mid x\langle \mathsf{return}(x)\rangle \mid \nu(x{:}T).\gamma \mid (x{:}T).\gamma$	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 5. Specification language

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. 3; 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 (together with its type T), 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 (of type T), either fresh or already known.

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. Hence, the input interactions are the ones being *scheduled,* whereas the outputs are not; they are used for *testing* that the object behaves correctly. 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, i.e., choices are either under control of the object itself and concerns outgoing communication, or under control of the environment and concerns incoming communication. These restrictions are formalized next as part of the well-formedness conditions.

3.1 Well-formedness

The grammar given in Tab. 5 allows to specify sets of traces. Not all specifications, however, are meaningful, i.e., describe traces actually possible at the interface of a component. We therefore formalize conditions to rule out such ill-formed specification where the main restrictions are: *Typing:* Values handed over must correspond to the expected types for that methods. *Scoping:* Variables must be declared (together with their types) before their use. *Communication patterns:* No value can be returned before a matching outgoing call has been seen at the interface. Specifications adhering to these restrictions are called *well-formed.*

Well-formedness is given straightforwardly by structural induction by the rules of Tab. 6. The rules formalize a judgment of the form

$$\Xi \vdash \varphi : wf^p \tag{2}$$

which stipulates φ's well-formedness under the assumption context Ξ. The meta-variable p (for polarity) stands for either ?, !, or ?!, where ?! indicates the polarity for an empty sequence or for a process variable, and ? and ! indicate well-formed input and output specifications respectively. As before, Ξ contains bindings from variables and class names to their types. The class names are considered as constants and also, the context Ξ will remain unchanged during the well-formedness derivation, since all classes are assumed to be known in advance and class names cannot be communicated. This is in contrast to the variables, which represent

$$\frac{}{\Xi \vdash \epsilon : wf^{?!}}\ \text{WF-EMTPY} \qquad \frac{\Xi \vdash X}{\Xi \vdash X : wf^{?!}}\ \text{WF-VAR}$$

$$\frac{a = \nu(\Xi').n\langle call\ o.l(\vec{v})\rangle? \qquad \Xi \vdash a : ok \qquad \acute{\Xi} = \Xi + a \qquad \acute{\Xi} \vdash \varphi : wf^p}{\Xi \vdash a.\varphi : wf^?}\ \text{WF-CALLI}$$

$$\frac{a = \nu(\Xi').n\langle return(v)\rangle? \qquad \Xi \vdash a : ok \qquad \acute{\Xi} = \Xi + a \qquad \acute{\Xi} \vdash \varphi : wf^p}{\Xi \vdash a.\varphi : wf^?}\ \text{WF-RETI}$$

$$\frac{\Xi \vdash \varphi_1 : wf^p \qquad \Xi \vdash \varphi_2 : wf^p}{\Xi \vdash \varphi_1 + \varphi_2 : wf^p}\ \text{WF-CHOICE} \qquad \frac{\Xi, X \vdash \varphi : wf^p}{\Xi \vdash \mathsf{rec}\ X.\varphi : wf^p}\ \text{WF-REC}$$

Table 6. Well-formedness of trace specifications

object references and references to future variables (resp. thread names). Besides that, the context also stores process variables X. The rules work as follows: The empty trace is well-formed (cf. rule WF-EMPTY), and a process variable X is well-formed, provided it had been declared before (written $\Xi \vdash X$, cf. rule WF-VAR). We omit the rules WF-CALLO and WF-RETO for outgoing calls, resp. outgoing get-labels, as they are dual to WF-CALLI and WF-RETI.

3.2 Observational blur

Creol objects communicate asynchronously and the order of messages might not be preserved during communication. Thus, an outside observer or tester can not see messages in the order in which they had been sent, and we need to relax the specification up-to some appropriate notion of *observational equivalence*, denoted by $\equiv_{obs}$ and defined by the rules of Tab. 7. Rule EQ-SWITCH captures the asynchronous nature of communication, in that the order of outgoing communication does not play a role. The definition corresponds to the one given in [29] and also of [18], in the context of multi-threading concurrency. Rule EQ-PLUS allows to distribute an output over a non-deterministic choice, *provided* that it's a choice itself over outputs, as required by the well-formed condition in the premise. Rule EQ-REQ finally expresses the standard unrolling of recursive definitions. We omit further standard equivalence rules, such as defining commutativity and associativity of $+$ and neutrality of ϵ.

Next we state that well-formedness is preserved under the given equivalence.

Lemma 1. *If $\Xi \vdash \varphi : wf^p$ and $\varphi \equiv_{obs} \varphi'$, then $\Xi \vdash \varphi' : wf^p$.*

Given the equivalence relation, the meaning of a specification is given operationally by the rather obvious reduction rules of Tab. 8. The next lemmas

$$\frac{}{\nu(\Xi).\gamma_1!.\gamma_2!.\varphi \equiv_{obs} \nu(\Xi).\gamma_2!.\gamma_1!.\varphi}\ \text{Eq-Switch} \qquad \frac{\vdash (\varphi_1 + \varphi_2) : wf^!}{\gamma!.(\varphi_1 + \varphi_2) \equiv_{obs} \gamma!.\varphi_1 + \gamma!.\varphi_2}\ \text{Eq-Plus}$$

$$\mathsf{rec}\ X.\varphi \equiv_{obs} \varphi[\mathsf{rec}\ X.\varphi/X] \qquad \text{Eq-Rec}$$

Table 7. Observational equivalence

$$\frac{\acute{\Xi} = \Xi + a}{\Xi \vdash a.\varphi \xrightarrow{a} \acute{\Xi} \vdash \varphi}\ \text{R-Pref} \qquad \frac{\Xi \vdash \varphi_1 \xrightarrow{a} \acute{\Xi} \vdash \varphi_1'}{\Xi \vdash \varphi_1 + \varphi_2 \xrightarrow{a} \acute{\Xi} \vdash \varphi_1'}\ \text{R-Plus}_1$$

$$\frac{\varphi \equiv_{obs} \varphi' \qquad \Xi \vdash \varphi' \xrightarrow{a} \Xi \vdash \varphi''}{\Xi \vdash \varphi \xrightarrow{a} \Xi \vdash \varphi''}\ \text{R-Equiv}$$

Table 8. φ rules

express simple properties of the well-formedness condition, connecting it to the reduction relation.

Lemma 2. *Assume* $\Xi \vdash \varphi : wf$.

1. *Exactly one of the three conditions holds:* $\Xi \vdash \varphi : wf^{?!}$, $\Xi \vdash \varphi : wf^?$, *or* $\Xi \vdash \varphi : wf^!$
2. *If* $\varphi \xrightarrow{a}$ *with* a *an input, then* $\Xi \vdash \varphi : wf^?$. *Dually for outputs.*
3. *If* $\Xi \vdash \varphi : wf^?$, *then* $\varphi \xrightarrow{a}$ *with* a *an input. Dually for outputs.*

Lemma 3 (Subject reduction). $\Xi \vdash \varphi : wf$ *and* $\Xi \vdash \varphi \xrightarrow{a} \acute{\Xi} \vdash \acute{\varphi}$, *then* $\acute{\Xi} \vdash \acute{\varphi} : wf$.

Lemma 4. *Assume* $\Xi \vdash C$. *If* $\Xi \vdash C \stackrel{t}{\Longrightarrow}$, *then* $\Xi \vdash \varphi_t : wf$ *(where* φ_t *is the trace* t *interpreted to conform to Tab. 5, i.e., the names of* t *are replaced by variables).*

4 Scheduling and asynchronous testing of Creol objects

Next we put together the (external) behavior of an object (Sect. 2) and its intended behavior specified as in Sect. 3. Table 9 defines the interaction of the interface description 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

$$\frac{\Xi \vdash C \xrightarrow{\tau} \Xi \vdash \acute{C}}{\Xi \vdash C \parallel \varphi \rightarrow \Xi \vdash \acute{C} \parallel \varphi}\ \textsc{Par-Int}$$

$$\frac{\Xi \vdash \varphi : wf^{?}}{\Xi \vdash \nu(\Xi').(C \parallel n\langle \mathsf{let}\ x{:}T = o.l(\vec{v})\ \mathsf{in}\ t\rangle \parallel \varphi) \rightarrow \lightning}\ \textsc{Par-Error}$$

$$\frac{\Xi_1 \vdash C \xrightarrow{a} \acute{\Xi}_1 \vdash \acute{C} \qquad \Xi_1 \vdash \varphi \xrightarrow{b} \acute{\Xi}_2 \vdash \acute{\varphi} \qquad \vdash a \lesssim_\sigma b}{\Xi_1 \vdash C \parallel \varphi \rightarrow \acute{\Xi}_1 \vdash \acute{C} \parallel \acute{\varphi}\sigma}\ \textsc{Par}$$

Table 9. Parallel composition

if it matches an outgoing label in the specification 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. The matching is not simple pattern matching as it needs to take into account in particular the two different kinds of bindings in the specification language, $\nu(x{:}T)$ as the freshness assertion and $(x{:}T)$ representing standard variable declarations . 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. We omit the details of the matching and refer to the longer version [15]. Rule Par-Error reports an error if the specification requires an input as next step and the object however could do an output. In the rule $\lightning$ indicates the ocurrence of an error. Note that the equivalence relation, according to rule Eq-Switch, allows the reordering of outputs, but not inputs.

Example 1. To illustrate the testing we sketch the well-known example of a travel agency. A client asks the travel agent for a cheap flight and the travel agent finds the cheapest flight by asking the flight companies. To test an implementation of the travel agent program we give a specification modeling the behavior of the client and the flight companies and specifying the expected behavior of the travel agent. The client sends two messages. First a start message and then the request. The travel agent tries to get the price information from the flight companies and then reports the result to the client.

$$\begin{aligned}\varphi_b = {} & n_{c1}\langle call\ b.start()\rangle?\ .\ n_{c1}\langle return()\rangle!\ .\ n_{c2}\langle call\ b.getPrice(x)\rangle?\ . \\ & n_1\langle call\ p_1.l(x)\rangle!\ .\ n_2\langle call\ p_2.l(x)\rangle!\ . \\ & n_1\langle return(v_1)\rangle?\ .\ n_2\langle return(v_2)\rangle?\ .\ n_{c2}\langle return(min_v)\rangle!\end{aligned}$$

5 Implementing a specification-driven Creol interpreter

The operational semantics of the object-oriented language Creol [20] is formalized in rewriting logic [22] and executable on the Maude rewriting engine [8]. To obtain a *specification-driven interpreter* for testing Creol objects, we have formalized our behavioral interface specification language in rewriting logic, too. In the combined implementation we synchronize communication between specification terms and objects. The specification generates the required input to the object and tests whether the output behaviour of the object conforms to the specification. The original Creol interpreter consists of 21 rewrite rules and the extension adds 20 more.

We have argued that specified method calls should not be placed into the callee's input queue, but the call should be answered immediately. I.e., if an incoming call is specified and the lock of the object is free, the corresponding method code should start executing immediately. In the current version of the interpreter the incoming messages are generated from the specification, which amounts to the same as only allowing scheduled calls to interact with the object.

A Creol state configuration is a multiset of objects, classes, and messages. The rewrite rules for state transitions are on the form `rl Cfg => Cfg'`, effectively evolving the state of one object by executing a statement. Some statements generate new messages. Finally, some rules are concerned with scheduling processes and receiving messages. For the scheduling 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: any interaction only takes place when it matches a complementary label in the specification. For example, the Par rule in Tab. 9 is implemented by several Maude rules, of which we show `Par-incoming-call` and `Par-remote-async-call`, that handle the cases of synchronized incoming and outgoing calls; we also show the `Par-Error` rule in Tab. 10. The rules are conditional rewrite rules, in which conditions of the form `Var:=term` bind `term` to the variable `Var`. Parts of the term that are not changed, like attributes, are represented by "...".

The rule `Par-incoming-call` combines the R-Pref rule in Tab. 8 for the specification with the CallI rule in Tab. 4 for interface behavior via the Par rule. The rule only applies if the process, `Pr` of the object `<O:C | ...>` is `idle` (i.e., the lock is free). The specification for `O`, `<call(T,R,M,P)? . sp>(O)`, starts with an incoming call label with thread name `T`, receiver `R`, method name `M`, and parameters `P`, and could by R-Pref reduce to `sp`. The careful reader might expect that the receiver mentioned in the specification should be the same as the object identifier `O`. However since a specification can contain variables, the receiver `R` *might* be identical to `O` but it may also be a variable, which will be matched with `O` in the `procLab` function. The function `procLab` (process label) generates concrete values from the variables in the specification label; builds an *invoc* message, i.e. a term representing a method call; and returns the message and a mapping of the variables to the values. The message and the

```
crl <call(T,R,M,P)?.sp>(O) || <O:C | ..., Pr:idle, ...> Cfg
 => <app(getS(Res),sp)>(O) || <O:C | ..., Pr:synch, ...> getM(Res) Cfg
if Res:=procLab(O , call(T,R,M,P)?) [label Par-incoming-call] .

crl <call(T,R,M,P)!.sp>(O) || <O:C | ...,Pr:{L | call(A;E;Q;EL);SL},...> Cfg
 => <app(Sub,sp)>(O)       || <O:C | ...,Pr:{insert(A,Lab,L) | SL},...> Cfg
    (invoc(O,Lab,Q,Args) from O to Rcv)
if Lab:=label(O,N) ∧ Rcv:=evalGuard(E,(S::L),noMsg) ∧
   Args:=evalGuardList(EL,(S::L),noMsg) ∧
   Sub:=matchCall(Lab,Rcv,Q,Args,call(T,R,M,P)) ∧ noMismatch(Sub)
[label Par-remote-async-call] .

crl <inSp>(O)    || <O:C | ...,Pr:{L | call(A;E;Q;EL);SL},...> Cfg
 => <epsilon>(O) || <O:C | ...,Pr:{L | call(A;E;Q;EL);SL},...> Cfg
    errorMsg("ERROR") if E=/="this" [label Par-Error] .
```

Table 10. Sample Maude rules

substitution are extracted by the functions `getM` and `getS`, respectively. The message is placed into the configuration and the substitution is applied to `sp` using the `app` function. Method binding and the rules for executing the bound code are specified by equations in the Creol interpreter. Since equations will be applied before any other rewrite rules this ensures that the execution of the code resulting from the call starts before any other invoc message can interfere.

In the `Par-remote-async-call` rule the object is in a state where the next step in the executing process is an outgoing call and the specification starts with a call out. The `matchCall` function tries to match the concrete values derived from the object's state against the variables in the label. The condition `noMismatch(Sub)` blocks the conditional rule if no match is possible, otherwise the outgoing call takes place and the substitution `Sub` is applied to the remainder of the specification. The last rule implements the Par-Error rule. The distinction between input and output specifications is enforced by different subsorts: the variable `inSp` matches all specifications of incoming messages. When the next step of the executing process is a call statement, then this leads to an error, as expected.

Here we focus on the run-time behavior of specifications. Hence, we simply assume well-formedness and don't give the Maude formalization.

6 Conclusion

We have presented a formalization of the interface behavior of Creol together with a behavioral interface specification language. We have formally described how to use this specification language for black-box testing of asynchronously communicating Creol components and we have presented our rewriting logic implementation of the testing framework.

Related work Systematic testing is indispensable to assure quality of software and systems (cf. [25,27,14,4,3], amongst others). [7] 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 (and as in [2] [12] [11] [13]). In the approach, pairs of (ground) terms are used for the test cases. Testing for *concurrent* object-oriented programs based on synchronization sequences is investigated in [6], using Petri nets and OBJ as foundation. Long in his thesis [21] 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. It can be seen as an extension of the testing method for monitors from [5]. For scheduling the intended order, an external *clock* is used, which is introduced for the purpose of testing, only. In the context of $C^{\#}$, [17] presents model-based analysis and model-based testing, where abstract models of object-oriented programs are tested. The approach, however, does not target concurrent programs.

Even if not specifically targeting Creol, [19] pursues similar goals as this paper, validating component interfaces specified in rewriting logic. In contrast to here, the interface behavior is specified by first-order logic over traces, where from a given predicate an assumption part and a guarantee part can be derived. Our approach is more specific in that we schedule incoming calls to a component, and test the output behavior. In [28], the authors target Creol as language and investigate how different schedulings of object activity restrict the behavior of a Creol object, thus leading to more specific test scenarios. The focus, however, is on the *intra*-object scheduling, and the test purposes are given as assertions on the *internal* state of the object. This is in contrast to the setting here, focusing on the interface communication. The testing methodologies are likewise different. We execute the behavioral trace specification directly in composition with the implementation being tested. They use a scheduling strategy and a model for an object implementation to generate test cases which then are used afterwards to test for compliance with an implemented Creol object.

Future work We plan to extend the theory to components under test instead of single objects. This leads to complex scheduling policies and complex specifications. Furthermore, there are several interesting features of the Creol language which may be added, including first-class futures, promises, processor release points, inheritance and dynamic class updates. For the specification language we want to investigate how to extend it with assertion statements on labels, which leads to scheduling policies sensitive to the *values* in the communication labels. Natural further steps for the implementation is to extend it to include a check for well-formedness according to Tab. 6, and also to modify the matching algorithm to distinguish between fresh and already known names . The generation of Creol messages from specifications can also be made more sophisticated to achieve better test coverage. It is also interesting to combine the approach we describe here with model checking and abstraction. By using the built-in *search*

functionality of Maude, model checking of invariants can be done easily. We plan to additionally use Maude's LTL model checker with our testing framework.

Acknowledgement We thank Andreas Grüner for giving insight to the field of testing of (concurrent) object-oriented languages, the members of the PMA group for valuable feedback and the anonymous referees for insightful and constructive criticism.

References

1. M. Abadi and L. Cardelli. *A Theory of Objects*. Monographs in Computer Science. Springer-Verlag, 1996.
2. G. Bernot, M.-C. Gaudel, and B. Marre. Software testing based on formal specifications. *IEEE Software Engineering Journal*, 6(6):387–405, Nov. 1991.
3. A. Bertolino. Software testing research: Achievements, challenges, dreams. In *Proceedings of Future of Software Engineering at ICSE 2007*, pages 85–103, May 2007.
4. R. V. Binder. *Testing Object-Oriented Systems, Models, Patterns, and Tools*. Addison-Wesley, 2000.
5. P. Brinch Hansen. Reproducible testing of monitors. *Software – Practice and Experience*, 8(223–245), 1978.
6. H. Y. Chen, Y. X. Sun, and T. H. Tse. A strategy for selecting synchronization sequences to test concurrent object-oriented software. In *Proceedings of the 27th International Computer Software and Application Conference (COMPSAC 2003), Los Angeles, California*. IEEE Computer Science Press, 2003.
7. 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.
8. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. The Maude 2.0 system. In Nieuwenhuis [26], pages 76–87.
9. The Creol language. `http://heim.ifi.uio.no/creol`, 2007.
10. W.-P. de Roever, F. S. de Boer, U. Hannemann, J. Hooman, Y. Lakhnech, M. Poel, and J. Zwiers. *Concurrency Verification: Introduction to Compositional and Noncompositional Proof Methods*. Cambridge University Press, 2001.
11. R.-K. Doong and P. G. Frankl. Case studies on testing object-oriented programs. In *Proceedings of the 4th Annual Symposium on Software Testing, Analysis, and Verification (TAF 4)*, pages 165–177, 1991.
12. R.-K. Doong and P. G. Frankl. The ASTOOT approach to testing object-oriented programs. *ACM Transactions on Software Engineering and Methodology*, 3(2):101–130, 1994.
13. P. G. Frankl and R.-K. Doong. Tools for testing object-oriented programs. In *Proceedings of the 8th Northwest Conference on Software Quality*, pages 309–324, 1990.
14. M.-C. Gaudel. Testing can be formal, too. In Mosses et al. [24], pages 82–96.
15. I. Grabe, M. Kyas, M. Steffen, and A. B. Torjusen. Executable interface specifications for testing asynchronous Creol components. Technical Report 375, University of Oslo, Dept. of Computer Science, July 2008. A shorter version has been submitted for conference proceedings.

16. IEEE. *Seventeenth Annual Symposium on Logic in Computer Science (LICS) (Copenhagen, Denmark)*. Computer Society Press, July 2002.
17. J. Jacky, M. Veanes, C. Campbell, and W. Schulte. *Model-Based Software Testing and Analysis with C♯*. Cambridge University Press, 2008.
18. A. Jeffrey and J. Rathke. A fully abstract may testing semantics for concurrent objects. In LICS'02 [16], pages 101–112.
19. E. B. Johnsen, O. Owe, and A. B. Torjusen. Validating behavioral component interfaces in rewriting logic. *Fundamenta Informaticae*, 82(4):341–359, 2008.
20. E. B. Johnsen, O. Owe, and I. C. Yu. Creol: A type-safe object-oriented model for distributed concurrent systems. *Theoretical Computer Science*, 365(1–2):23–66, Nov. 2006.
21. B. Long. *Testing Concurrent Java Components*. PhD thesis, University of Queensland, July 2005.
22. J. Meseguer. Conditional rewriting as a unified model of concurrency. *Theoretical Computer Science*, pages 73–155, 1992.
23. R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, part I/II. *Information and Computation*, 100:1–77, Sept. 1992.
24. P. D. Mosses, M. Nielsen, and M. I. Schwarzbach, editors. *TAPSOFT '95: Theory and Practice of Software Development, 6th International Joint Conference CAAP/FASE*, volume 915 of *LNCS*. Springer, 1995.
25. G. J. Myers. *The Art of Software-Testing*. John Wiley & Sons, New York, 1979.
26. R. Nieuwenhuis, editor. *Proceedings of the 14th International Conference on Rewriting Techniques and Applications (RTA 2003)*, volume 2706 of *LNCS*. Springer, June 2003.
27. R. Patton. *Software Testing*. SAMS, second edition, July 2005.
28. R. Schlatte, B. Aichernig, F. de Boer, A. Griesmayer, and E. B. Johnsen. Testing concurrent objects with application-specific schedulers. In J. Fitzgerald and A. Haxthausen, editors, *International Colloquium on Theoretical Aspects of Computing (ICTAC)*, volume 5160 of *LNCS*. Springer, 2008.
29. M. Steffen. *Object-Connectivity and Observability for Class-Based, Object-Oriented Languages*. Habilitation thesis, Technische Faktultät der Christian-Albrechts-Universität zu Kiel, July 2006.