UNIVERSITY OF OSLO
Department of Informatics

Executable interface specifications for testing asynchronous Creol components

Research Report 375

Immo Grabe

Martin Steffen

Arild B. Torjusen

ISBN 82-7368-335-4
ISSN 0806-3036

14. July 2008

Executable interface specifications for testing asynchronous Creol components*

14. July 2008

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

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

Abstract. Creol is a high-level, object-oriented language for distributed systems, featuring active objects and asynchronous method calls. In this paper we present a behavioral interface specification language over communication trace labels to specify components in terms of traces of observable behavior.
In the specification, a clean separation of concerns between interaction under the 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 order of inputs, whereas the outputs as commitments are being tested for conformance. To ensure the mentioned separation of responsibilities, we define well-formedness conditions which in addition assure that only "meaningful" traces, i.e., those corresponding to actual behavior, can be specified. The specification language is characterized by two other salient features: it allows to specify freshness of communicated values and furthermore, it respects the asynchronous nature of communication in Creol: the output is tested only up-to an appropriate notion of observability.

1 Introduction

Reasoning about open distributed systems and predicting their behavior is intrinsically difficult and one reason for that is their inherent asynchronicity and the resulting non-determinism. It is generally accepted that the only viable 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 [19], a programming and modeling language for distributed systems based on concurrent, active objects communicating via asynchronous method calls.

* 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).

To describe and test Creol components, we introduce a simple 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 of the environment, input is considered as assumption about the environment whereas output describes commitments of the object. For input interactions, we will 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 will *test* that the events occur as specified. An expression in the specification language thus gives an assumption-commitment style specification 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 compliance of an implementation of a component with the interface specification, where incorrect or noncomplying behavior of the component under a given scheduling is reported as an error.

As mentioned, 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 compliance. We do so by formalizing *well-formedness* conditions on specifications. 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. Secondly, it reflects the asynchronous nature of the communication model: the order in which messages are sent is not necessarily the order in which they arrive at an external observer. Hence, in other words, the trace specifications are considered only up-to an appropriate notion of *observable equivalence*, taking especially the asynchronous message passing into account.

Overview The paper is organized as follows. In Section 2 we introduce a calculus capturing core aspects of the Creol language and Section 3 presents the behavioral specification language and formulates the well-formedness conditions. Section 4 explains how to compose a Creol program and a specification and how to use them for testing. Section 5 finally concludes with related and future work.

2 The Creol language

Creol [8,19] is a high-level object-oriented language for distributed systems, featuring active objects and asynchronous method calls. Concentrating on the core

features, we elide first-class futures (recently introduced in *Creol* [9]), 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* (i.e., the term input "queue" is a slight misnomer, as it seems to indicate fifo-discipline in the scheduling).

We start with the abstract syntax in Section 2.1. Afterwards, Section 2.2 contains the startic typing rule and Section 2.3 the operational semantics.

2.1 Syntax

The abstract syntax of the calculus is given in Table 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 latter are not found in a program written by the user, but generated at run-time by the rules of the operational semantics.

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

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 Table 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[3], 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 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". Incarnations insofar, as the formal parameters have been replaced by actual ones, especially the method's self-parameter has been replaced by the identity of the target object of the method call. The term t is basically a sequence of expressions, where the let-construct is used for sequencing and for local declarations.[4] During execution, $n\langle t\rangle$ contains in t the running code of a method body. When evaluated, the thread is of the form $n\langle v\rangle$ and the value can be accessed via n, the future reference, or future for short.

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. In the terminology of Java, all methods are implicitly considered "synchronized". The crucial difference between Java's multi-threading concurrency model and Creol's active objects model used here is the way method calls are issued at the

[3] There will be no result value in case of non-terminating methods.

[4] $t_1; t_2$ (sequential composition) abbreviates $\mathsf{let}\, x{:}T = t_1\, \mathsf{in}\, t_2$, where x does not occur free in t_2.

caller site. In Java and similar languages, method calls are *synchronous* in the sense that the calling activity blocks to wait for the return of the result and thus the control is transferred to the callee. 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. In that way, a method call never transfers control from one object, the caller, to another one, the callee. In other words, no thread ever crosses the boundaries of an object, which means, the boundaries of an object are at the same time boundaries of the threads and thus, the objects are at the same time units of concurrency. Thus, the objects are harnessing the activities and can be considered as bearers of the activities. This is typical for object-oriented languages based on *active objects*.

The ν-operator is used for hiding and dynamic scoping, as known from the π-calculus [21]. 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 (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, of course. 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. 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. Note that the syntax does not allow to set a field back to undefined. Direct access (read or write) 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. As mentioned, objects come equipped with binary locks, responsible for assuring mutual exclusion. The two basic, complementary operations on a lock are grab and release. The first allows an activity to acquire access in case the lock is free ($\bot$), thereby setting it to $\top$, and release(o) conversely relinquishes the lock of the object o, giving other threads the chance to be executed in its stead, when succeeding to grab the lock via grab(o). The user is not allowed to directly ma-

nipulate the object locks. Thus, both expressions belong to the run-time syntax, underlined in Table 1, and are only generated and handled by the operational semantics as auxiliary expression at run-time. Instead of using directly grab and release, the lock-handling is done automatically when executing a method body: before starting to execute, the lock has to be acquired and upon termination, the lock is released again. Besides that, lock-handling is involved also when results are claimed, i.e., when a client code executing in an object, say o, intends to read the result of a future. The expression $\mathsf{claim}@(n, o)$ is the attempt to obtain the result of a method call from the future n while in possession of the lock of object o. There are two possibilities in that situation: either the value of the future has already been determined, i.e., the method calculating the result has terminated, in which case the client just obtains the value *without* loosing its own lock. In the alternative case, where the value is not yet determined, the client trying to read the value gives up its lock via release and continues executing only after the requested value has been determined (using get to read it) and after it has re-acquired the lock. Unlike claim, the get-operation is not part of the user-syntax. Both expressions are used to read back the value from a future and the difference in behavior is that get unconditionally attempts to get the value, i.e., blocks until the value has arrived, whereas claim gives up the lock temporarily, if the value has not yet arrived, as explained. Note the order in which get and grab are executed after releasing the lock: the value is read in via get *before* the lock has actually been re-acquired! We assume by convention, that when appearing in methods of classes, the claim- and the suspend-command only refer to the self-parameter *self*, i.e., they are written $\mathsf{claim}@(n, \mathit{self})$ and $\mathsf{suspend}(\mathit{self})$.

2.2 Typing

The calculus is strongly typed. The static type system is rather conventional and coincides largely with the one given in [26]. We included the rules, for the sake of completeness, in the appendix of this technical report, in Tables 13 and 14, without much explanations. The typing jugdgments are of the following form: $\Delta \vdash C : \Theta$ on the level of components (cf. Table 13) asserts well-typedness of C under the assumption name context Δ and with commitments Θ. On the level of threads, expressions, and their sub-phrases, $\Gamma; \Delta \vdash t : T$ asserts well-typedness of thread t with type T, under the name assumptions Δ and variable assumptions Γ.

2.3 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 Table 2 and 5. The internal rules of Table 2 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 fairly straightforward, and we show them for reference only. The steps are given as unlabelled steps, where we distinguish between $\rightsquigarrow$-steps (confluent) and

$\xrightarrow{\tau}$-steps (non-confluent, accessing the instance state) If the distinction does not play a role, we write $\rightarrow$. Components and the reduction relation are interpreted up-to standard structural congruences (cf. Table 15 and 16 in the appendix). We write $\Longrightarrow$ for the reflexive and transitive closure of the internal steps from Table 16.

$$n\langle \mathsf{let}\, x{:}T = v\, \mathsf{in}\, t\rangle \rightsquigarrow n\langle t[v/x]\rangle \qquad \textsc{Red}$$

$$n\langle \mathsf{let}\, x_2{:}T_2 = (\mathsf{let}\, x_1{:}T_1 = e_1\, \mathsf{in}\, e)\, \mathsf{in}\, t\rangle \rightsquigarrow n\langle \mathsf{let}\, x_1{:}T_1 = e_1\, \mathsf{in}\ (\mathsf{let}\, x_2{:}T_2 = e\, \mathsf{in}\, t)\rangle \qquad \textsc{Let}$$

$$n\langle \mathsf{let}\, x{:}T = (\mathsf{if}\, v = v\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2)\, \mathsf{in}\, t\rangle \rightsquigarrow n\langle \mathsf{let}\, x{:}T = e_1\, \mathsf{in}\, t\rangle \qquad \textsc{Cond}_1$$

$$n\langle \mathsf{let}\, x{:}T = (\mathsf{if}\, v_1 = v_2\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2)\, \mathsf{in}\, t\rangle \rightsquigarrow n\langle \mathsf{let}\, x{:}T = e_2\, \mathsf{in}\, t\rangle \qquad \textsc{Cond}_2$$

$$n\langle \mathsf{let}\, x{:}T = (\mathsf{if}\, \mathsf{undef}(\bot_{c'})\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2)\, \mathsf{in}\, t\rangle \rightsquigarrow n\langle \mathsf{let}\, x{:}T = e_1\, \mathsf{in}\, t\rangle \qquad \textsc{Cond}_1^{\bot}$$

$$n\langle \mathsf{let}\, x{:}T = (\mathsf{if}\, \mathsf{undef}(v)\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2)\, \mathsf{in}\, t\rangle \rightsquigarrow n\langle \mathsf{let}\, x{:}T = e_2\, \mathsf{in}\, t\rangle \qquad \textsc{Cond}_2^{\bot}$$

$$n\langle \mathsf{let}\, x{:}T = \mathsf{stop}\, \mathsf{in}\, t\rangle \rightsquigarrow n\langle \mathsf{stop}\rangle \qquad \textsc{Stop}$$

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

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

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

$$n_1\langle v\rangle \parallel n_2\langle \mathsf{let}\, x : T = \mathsf{claim}@(n_1, o)\, \mathsf{in}\, t\rangle \rightsquigarrow n_1\langle v\rangle \parallel n_2\langle \mathsf{let}\, x : T = v\, \mathsf{in}\, t\rangle \qquad \textsc{Claim}_i^1$$

$$\frac{t_2 \neq v}{n_2\langle t_2\rangle \parallel n_1\langle \mathsf{let}\, x : T = \mathsf{claim}@(n_2, o)\, \mathsf{in}\, t_1'\rangle \rightsquigarrow \quad n_2\langle t_2\rangle \parallel n_1\langle \mathsf{let}\, x : T = \mathsf{release}(o); \mathsf{get}@n_2\, \mathsf{in}\, \mathsf{grab}(o); t_1'\rangle} \textsc{Claim}_i^2$$

$$n_1\langle v\rangle \parallel n_2\langle \mathsf{let}\, x : T = \mathsf{get}@n_1\, \mathsf{in}\, t\rangle \rightsquigarrow n_1\langle v\rangle \parallel n_2\langle \mathsf{let}\, x : T = v\, \mathsf{in}\, t\rangle \qquad \textsc{Get}_i$$

$$n\langle \mathsf{suspend}(o); t\rangle \rightsquigarrow n\langle \mathsf{release}(o); \mathsf{grab}(o); t\rangle \qquad \textsc{Suspend}$$

$$o[c, F, \bot] \parallel n\langle \mathsf{grab}(o); t\rangle \xrightarrow{\tau} o[c, F, \top] \parallel n\langle t\rangle \qquad \textsc{Grab}$$

$$o[c, F, \top] \parallel n\langle \mathsf{release}(o); t\rangle \xrightarrow{\tau} o[c, F, \bot] \parallel n\langle t\rangle \qquad \textsc{Release}$$

Table 2. Internal steps

The communication labels, the basic building blocks of the interface interactions, are given in Table 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 $n\langle \mathit{call}\ o.l(\vec{v})\rangle$ 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 $n\langle \mathit{return}(v)\rangle?$ 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

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

Table 3. Communication labels

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 (whose names are assumed all disjoint, as usual) 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 $\mathit{fn}(a)$ and the bound names $\mathit{bn}(a)$ of a label a are defined as usual, whereas $\mathit{names}(a)$ refer to all names of a.

The interface behavior is given by the 4 rules of Table 5, which correspond to the 4 different kinds of labels, a call or a return, either incoming or outgoing. 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. An important, but standard, part of the external semantics is to check the static *typing* assumptions, e.g., whether at most the names actually occurring in the core of the label are mentioned in the ν-binders of the label and whether the transmitted values are of the correct types. Besides *checking* whether the assumptions are met before a transition, the contexts are *updated* by a transition step. 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 Table 5. 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 $\Xi \vdash a : \mathit{ok}$, instead. The right-hand notation of (1) extends the binding context Ξ by the bindings transmitted as part of label a appropriately. The formal definition of context update is given below and checking of static typing assumptions is defined as follows:

Definition 1 (Well-formedness and well-typedness). *A label $a = \nu(\Xi).\lfloor a\rfloor$ is* well-formed, *written $\vdash a$, if $\mathit{dom}(\Xi) \subseteq \mathit{names}(\lfloor a\rfloor)$ and if Ξ is a well-formed name-context for object and future names, i.e., no name bound in Ξ occurs twice. The assertion*

$$\acute{\Xi} \vdash o.l? : \vec{T} \to T \tag{2}$$

$$\frac{\acute{\Xi} \vdash n : [T] \quad ; \acute{\Xi} \vdash \vec{v} : \vec{T} \quad a = n\langle call\ o.l(\vec{v})\rangle?}{\acute{\Xi} \vdash a : \ \vec{T} \to \_} \text{ LT-CALLI}$$

$$\frac{; \acute{\Xi} \vdash v : T \quad a = n\langle return(v)\rangle?}{\acute{\Xi} \vdash a : \ \_ \to T} \text{ LT-RETI}$$

Table 4. Typechecking labels

("an incoming call of the method labeled l in object o expects arguments of type $\vec{T}$ and results in a value of type T") is given by the following rule, i.e., implication:

$$\frac{; \acute{\Theta} \vdash o : c \qquad ; \acute{\Xi} \vdash c : [\![\ldots, l{:}\vec{T} \to T, \ldots]\!]}{\acute{\Xi} \vdash o.l? : \vec{T} \to T} \tag{3}$$

For outgoing calls, $\acute{\Xi} \vdash o.l! : \vec{T} \to T$ is defined dually. In particular, in the first premise, $\acute{\Theta}$ is replaced by $\acute{\Delta}$. Well-typedness *of an incoming core label a with expected type $\vec{T}$, resp., T, and relative to the name context $\acute{\Xi}$ is asserted by*

$$\acute{\Xi} \vdash a : \vec{T} \to \_ \quad \textit{resp.}, \quad \acute{\Xi} \vdash a : \_ \to T \ , \tag{4}$$

as given by Table 4.

Note that the receiver o of the call is checked using only the commitment context $\acute{\Theta}$, to assure that o is a component object. Note further that to check the interface type of the class c, the full $\acute{\Xi}$ is consulted, since the argument types $\vec{T}$ or the result type T may refer to both component and environment classes.

As mentioned, the contexts are *updated* by a transition step. especially they are extended by the new names whose scope is extruded. For the binding part Ξ' of a label $\nu(\Xi').\gamma$, the scope of the references to existing objects and thread names Δ' extrudes across the border. In the step, Δ' extends the assumption context Δ and Θ' the commitment context Θ. This gives rise to the following definition.

Definition 2 (Context update). *Let Ξ be a name context and $a = \nu(\Xi').\lfloor a \rfloor$ an* incoming *label. Then the definitions of the post-contexts $\acute{\Delta}$ and $\acute{\Theta}$ are given as follows, when $n{:}[T]$ is the binding for the thread name:*

$$\acute{\Delta} = (\Delta, \Xi') \setminus n{:}T \quad \textit{and} \quad \acute{\Theta} = \Theta, n{:}T \ . \tag{5}$$

We write $\Xi + a$ for that update. For outgoing *communication, the definition is applied dually.*

Given the definitions for well-typedness and context update, we describe the rules of Table 5. 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 $o.l(\vec{v})$. 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 CallO_i from Table 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.

A trace of a well-typed component is a sequence of external steps; we write $\Xi_1 \vdash C_1 \stackrel{t}{\Longrightarrow} \Xi_2 \vdash C_2$ when the component $\Xi_1 \vdash C_1$ evolves to $\Xi_2 \vdash C_2$ by executing the trace t. The corresponding rules are given in Table 17. For $\Xi_1 \vdash C_1 \stackrel{\epsilon}{\Longrightarrow} \Xi_2 \vdash C_2$, we write shorter $\Xi_1 \vdash C_1 \Longrightarrow \Xi_2 \vdash C_2$, where ϵ denote the empty trace.

Remark 1. The rules for external steps from Table 5 resemble the ones given in [1]. There two differences are as follows. As we decided not to consider first-class futures and promises here (as in [1]), the set of rules is simpler here; rules dealing with obtaining the result of a future across the interface are not needed here. The second difference concerns the treatment of incoming calls in rule CallI. Here, an incoming call crossing the interface *atomically* grabs the lock, as we intend to describe and schedule the behavior and order the message communication in the order they are executed in the object. Thus, the object's input queue is not modeled here. This is in contrast to the formalization in [1]. □

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

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

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

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

Table 5. 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 Table 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. The syntax of the labels in the specification language, naturally, quite resembles the labels of Table 3. Comparing Tables 3 and 6, there are two differences: firstly, instead of names or references n, the specification language here uses variables. Secondly. the labels here allow a binding of the form $(x{:}T).\gamma$, which has no analog in Table 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.

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

Table 6. Communication labels

To specify sets of label traces, we employ a simple recursive trace language. Table 7 contains its syntax, describing sets of finite traces over labels.

$\varphi ::= X \mid \epsilon \mid a.\varphi \mid \varphi + \varphi \mid \mathsf{rec}\ X.\varphi$	specifications

Table 7. Specification language

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. Consequently, 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 as part of the well-formedness conditions in the next section.

Example 1. We give an example to illustrate the scoping.

$$\nu(n_1{:}[T_1])\nu(o_2{:}c_2)n_1\langle \mathit{call}\ o_2.l()\rangle!\ .\ \nu(n_2{:}[T_2])(o_3{:}c_2)n_2\langle \mathit{call}\ o_1.l'(o_3)\rangle?$$

The specification begins with o_1 calling method l of o_2. As being the first step both objects o_1 and o_2 and the future reference n_1 are new. After the initial call we expect a call from o_2 to o_1. This call will be made by the new thread n_2. We expect o_3 to be the parameter of this call. Since o_3 is given as variable it might be either new or old. □

The programming language Creol is strongly typed. Accordingly, also the interface specification language sets value on the fact to allow only specifications that "make sense" type-wise. It makes, e.g., no sense to specify traces that insist on transmitting values in method calls that do not fit to the expected values as declared in the type of the corresponding method. Such specifications are rejected as being ill-typed and the restriction is justified by the fact that no component (which is assumed to be well-typed) can produce an ill-typed trace. This fact will later be proved. Indeed, well-typedness is an important part of the general well-formedness conditions we impose on the specifications. The typing conditions are rather standard and we mostly elide the rules for typing. They resemble closely the ones of [26] and especially from [1] for legality of traces. The difference to the first case is that [26] deals with a calculus for Java instead of Creol, and the latter [1] is more complex than the typing as considered here, as it deals additionally with the concept of first-class futures and promises. In general, the close resemblance wrt. typing is not surprising: the earlier papers dealt with the interface behavior in forms of sets of traces, which is here generalized to a more expressive recursive language to specify such behavior.

Remark 2. As mentioned, an important distinction in the trace specification is the one between incoming communication and outgoing. Especially, we do not allow *mixed choices,* i.e., a choice $\varphi_1 + \varphi_2$ where, e.g., φ_1 starts with an input and φ_2 with an output. This distinction could be enforced syntactically, for instance by distinguishing syntactically between external and internal choices. Such a syntactic distinction is often found for instance in formalizations based on *session types* [15,27], which can be seen a behavioral, trace-based interface description formulated by type system (therefore the term "session type" and not "session trace" ...). Here we do not, however, reflect the distinction in the grammar of Table 7. Instead, the well-formedness conditions later discriminate between traces that start with an incoming communications and those starting with an output. □

3.1 Well-formedness

The grammar given in Table 7 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 specifications. The main restrictions are the following:

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 Table 8. The rules formalize a judgment of the form

$$\Xi \vdash \varphi : wf^p \tag{6}$$

which stipulates φ's well-formedness under the assumption context Ξ. The meta-variable p (for polarity) stands for either ?, !, or ?!. 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 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 two rules WF-CallO and WF-RetO dealing with outgoing calls, resp. outgoing get-labels, as they are dual to WF-CallI and WF-RetI.

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

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

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

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

Table 8. Well-formedness of trace specifications

Remark 3 (Regular expressions). The specification language as given in Table 7 uses label-prefixing to express sequentiality in a trace and recursion to represent infinite behavior. Specifications thus resemble a automata-like or process-algebra representation. An alternative design is to specify sets of traces using the syntax of regular languages, i.e., to allow sequential composition $\varphi_1;\varphi_2$ of two formulas and to use iteration φ^* for infinite behavior.

Using regular expressions as sketched would slightly complicate the formulation of the well-formedness conditions. To accommodated for general sequential composition (in contrast to simple prefixing), the judgment for well-formedness would need to mention also the context *after* the traces given by the formula. I.e., the well-formedness judgment of equation (6) would have to be generalized to

$$\Xi \vdash \varphi : wf^p :: \acute{\Xi} \ , \tag{7}$$

where $\acute{\Xi}$ is the mentioned context after φ. Besides that, care must be taken wrt. *scope* of the variables. For instance, in $(\varphi_1 + \varphi_2);\varphi_3$, the trailing φ_3 may only use variables that have been introduced *both* in φ_1 and φ_2. In other words, the scope of a variable introduced in φ_1, say, does not extend unconditionally to φ_2. In a similar spirit and given $\varphi_1^+;\varphi_2$ scope of variables introduced in φ_1 *ends* at then end of φ_1 and does not extend to φ_2.

Both representations, the one of Table 7 and the one sketched here based on regular expressions, are equally expressive. For the sake of simplicity for the formalization, especially concerning the well-formedness conditions, we base this paper on the one using label-prefixing and explicit recursion. In the examples we sometimes use the regular expression syntax instead. □

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*. This notion is defined by the rules of Table 9. 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 [26] and also of [17], 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. Eq-Plus-Comm expresses commutativity of choice.

The next lemma states that well-formedness is preserved under the given equivalence.

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

Proof. By induction on the rules of Table 8. Note that rule Eq-Plus explicitly requires output well-formedness of $\varphi_1 + \varphi_2$ it its premise. □

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

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

$$\varphi_1 + \varphi_2 \equiv_{obs} \varphi_2 + \varphi_1 \qquad \text{Eq-Plus-Comm} \qquad\qquad \varphi + \epsilon \equiv_{obs} \varphi \qquad \text{Eq-Plus-Empty}$$

Table 9. Equivalence

Given the equivalence relation, the meaning of a specification is given operationally, by the rather obvious reduction rules of Table 10.

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

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

Table 10. φ rules

The next lemmas 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.*

Proof. Part 1 by straightforward induction on the rules of Table 8. Part 2 by inverting the rules of 10 and by inspection of the rules for well-formedness. Part 3 works similarly.

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

Proof. By straightforward induction on derivations of the rules of Table 10. □

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 Table 7, i.e., the names of t are replaced by variables).*

Proof. By straightforward induction. The proof works similar to the proof in [26], which shows that the behavior of a component is a legal trace. Note in this context that φ_t is of a restricted form: it is constructed by prefixing and the empty trace, only. □

4 Scheduling and asynchronous testing of Creol objects

Next we put together the (external) behavior of an object (Section 2) and its intended behavior specified as in Section 3. Table 11 defines the interaction of the interface description with the component, basically by synchronous parallel composition: both φ and component must engage in corresponding steps, which, for incoming communication schedules the order of interactions with the component whereas for outgoing communication, will raise an error.

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

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

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

Table 11. Parallel composition

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; see Definition 3 below. Rule Par-Error finally reports an error, if the specification requires an input as next step, the object however could do an output. Note that the equivalence relation, according to rule Eq-Switch, allows the reordering of outputs, but not inputs.

Definition 3 (Matching). *Given two labels a_1 and a_2, we write $\vdash a_1 \lesssim_\sigma a_2$ (read "a_1 matches a_2 with substitution σ"), if that judgment can be derived by the rules of Table 12.*

The rules of Table 12 work as follows. They define the matching relation between two labels (written $\vdash a_1 \lesssim_\sigma a_2$), where a_1 is the label produced by the

$$\frac{}{\vdash () \lesssim () : ok}\ \text{M-Empty} \qquad \frac{\vdash \Xi_1 \lesssim \Xi_2 : ok}{\vdash \nu(n{:}T), \Xi_1 \lesssim \nu(n{:}T), \Xi_2 : ok}\ \text{M-NDec}$$

$$\frac{\vdash \Xi_1 \lesssim \Xi_2 : ok}{\vdash \nu(n{:}T), \Xi_1 \lesssim (n{:}T), \Xi_2 : ok}\ \text{M-Dec}_1 \qquad \frac{\vdash \Xi_1 \lesssim \Xi_2 : ok}{\vdash \Xi_1 \lesssim (n{:}T), \Xi_2 : ok}\ \text{M-Dec}_2$$

$$\frac{\vdash a_1 \lesssim_\sigma a_2 : ok \qquad \vdash \Xi_1\sigma \lesssim \Xi_2 : ok}{\vdash \Xi_1.a_1 \lesssim_\sigma \Xi_2.a_2 : ok}\ \text{M-Lab}$$

Table 12. Matching of labels

component and a_2 the one specified by the interface description. The subscript σ is the substitution, a mapping from variables to names, that gives rise to the match.

The difference between the two syntactic categories therefore is that a_2 may contain variables and a_1 may not, and furthermore, the grammar for a_2 allows bindings of the form $(x{:}T)$ and $\nu(x{:}T)$, whereas the first variant does not occur for labels a_1. A label a (for both cases) consists of a binding part and the core of the label without bindings. In slight abuse of notation, we write Ξ for the binding part or context. In rule M-Lab for matching the two labels, Ξ_1 thus contains bindings of the form $\nu(n{:}T)$, i.e., from names to types, denoting the *new* names exchanged with the object in that step. Ξ_2's bindings on the other hand, associating variables with types, are of the form $(x{:}T)$ or $\nu(x{:}T)$. The label $\Xi_1.a_1$ is matched against $\Xi_2.a_2$, as given by M-Lab in two steps. First, the cores a_1 and a_2 of the two labels are matched against each other by standard pattern matching, written $\vdash a_1 \lesssim_\sigma a_2$. In other words, $a_1 = a_2\sigma$, where σ is the (uniquely determined) substitution, which, when applied to a_2, gives a_1.[5]

The outer binding parts Ξ_1 and Ξ_2 are checked afterwards, as specified in the remaining 4 rules, where the variables of Ξ_1 are replaced by names, as given by the matching substitution σ. Note that the well-formedness conditions assure, that $\Xi_1\sigma$ no longer contains variable bindings, only bindings from names to types. Note further that we consider the contexts Ξ_1 and Ξ_2 as un-ordered, i.e., writing e.g. $(o{:}T), \Xi$ does not indicate that the binding $(o{:}T)$ occurs left-most in Ξ. In other words, the contexts Ξ, as usual, are understood up-to re-ordering. Rule M-NDec stipulates that if the specification requires a new name, the transmitted name must indeed be fresh. If, however, the specification just introduces a variable without insisting on freshness, then either the name can be fresh (cf. rule M-Dec$_1$) or the name had already been introduced, in which case it is ignored (cf. rules M-Dec$_2$). Finally, two empty contexts clearly match (cf. rule M-Empty).

[5] As an aside: assuming that both labels are checked for well-formedness, a type-mismatch between the name and the variable does not occur.

Example 2. 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 an initiation 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.

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

5 Conclusion

Related work Systematic testing is indispensable to assure quality of software and systems (cf. [23,24,14,4,3], amongst others). [7] present a approach to integrate black-box and white-box testing for object-oriented programs. Also based on the notion of observable equivalence, as introduced by [12] [11] for testing. 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 [20] presents ConAn ("concurrency analyser"), which generates test drivers from test scripts. The method allows to specifiy 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.

Even if not specifically targeting Creol, [18] 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. The assumption part of the specification is used to generate arbitrary input to the component under test, while the guarantee part is used for testing that the output from the component conforms to the given predicate. Our approach is more specific in that we *schedule* incoming calls to a component, and test the output behavior. Our specification language is not first-order logic but a recursive language over communication labels. As this paper, [25] targets Creol as language and investigates 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 their 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 intend to use the formalization presented here as specification of an implementation in rewriting logic, using Maude as execution platform and the already implemented framework of [18] as starting point. There are two different approaches to provide an interpreter for our theory. On the one hand the interpreter can be adapted to accept trace specifications as well as Creol program code and *execute* both. On the other hand the interpreter can be extended to accept scheduling policies and a Creol program and such a scheduling policy can be derived from the specification. Such an interpreter could be integrated into the already existing validation tool suite for Creol. Apart from the fact that the approach here is tailor-made for Creol (and not general for describing interface behavior), we expect a gain in efficiency. We see two reasons justifying that hope: First, the trace specification language is much simpler here, and thus more efficiently executable by rewriting. Secondly, the theory presented here took care to capture the effect of asynchronous message passing by an appropriate observational equivalence relation. Since Maude allows to directly specify behavior using reduction relation *modulo* equivalence, this will allow a more efficient treatment of the asynchronous behavior. Besides that, we plan to extend the theory to components under test instead of single objects. This leads to to complex scheduling policies and complex specifications due to the asynchronous and concurrent setting. 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. It might also be useful to apply model checking and abstraction to check conformance of an object to its interface specification.

Acknowledgement We thank Marcel Kyas for helpful discussions, e.g., about the intricacies of Creol, Andreas Grüner for giving insight to the field of testing of (concurrent) object-oriented languages, and Jasmin C.Blanchette for his remarks after reading a preliminary version of this report.

References

1. E. Ábrahám, I. Grabe, A. Grüner, and M. Steffen. Abstract interface behavior of an object-oriented language with futures and promises. Feb. 2008. Submitted as invited journal contribution to a special edition of the Journal of Logic and Algebraic Programming (NWPT'07). The paper is a reworked version of an earlier UiO Technical Report TR-364, Oct. 2007.
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. The Creol language. `http:heim.ifi.uio.no/creol`, 2007.
9. F. S. de Boer, D. Clarke, and E. B. Johnsen. A complete guide to the future. In de Nicola [10], pages 316–330.
10. R. de Nicola, editor. *ESOP'07*, volume 4421 of *Lecture Notes in Computer Science*. Springer-Verlag, 2007.
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. [22], pages 82–96.
15. K. Honda. Types for dyadic interaction. In E. Best, editor, *Proceedings of CONCUR '93*, volume 715 of *Lecture Notes in Computer Science*, pages 509–523. Springer-Verlag, 1993.
16. IEEE. *Seventeenth Annual Symposium on Logic in Computer Science (LICS) (Copenhagen, Denmark)*. Computer Society Press, July 2002.
17. A. Jeffrey and J. Rathke. A fully abstract may testing semantics for concurrent objects. In LICS'02 [16].
18. E. B. Johnsen, O. Owe, and A. B. Torjusen. Validating behavioral component interfaces in rewriting logic. *Fundamenta Informaticae*, 82(4):341–359, 2008.
19. 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.
20. B. Long. *Testing Concurrent Java Components*. PhD thesis, University of Queensland, July 2005.
21. R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, part I/II. *Information and Computation*, 100:1–77, Sept. 1992.
22. 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 *Lecture Notes in Computer Science*. Springer-Verlag, 1995.
23. G. J. Myers. *The Art of Software-Testing*. John Wiley & Sons, New York, 1979.
24. R. Patton. *Software Testing*. SAMS, second edition, July 2005.
25. R. Schlatte, B. Aichernig, F. de Boer, A. Griesmayer, and E. B. Johnsen. Testing (with) application-specific schedulers for concurrent objects. 2008. Accepted for ICTAC 2008, 5th International Colloquium on Theoretical Aspects of Computing.

$$\frac{}{\Delta \vdash \mathbf{0} : ()}\ \text{T-Empty} \qquad \frac{\Delta, \Theta_2 \vdash C_1 : \Theta_1 \quad \Delta, \Theta_1 \vdash C_2 : \Theta_2}{\Delta \vdash C_1 \parallel C_2 : \Theta_1, \Theta_2}\ \text{T-Par} \qquad \frac{\Delta \vdash C : \Theta, n{:}T}{\Delta \vdash \nu(n{:}T).C : \Theta}\ \text{T-Nu}$$

$$\frac{;\, \Delta, c{:}T \vdash [\![O]\!] : T}{\Delta \vdash c[\![O]\!] : (c{:}T)}\ \text{T-NClass} \qquad \frac{;\, \Delta \vdash c : [\![T_F, T_M]\!] \quad ;\, \Delta, o{:}c \vdash [F] : [T_F]}{\Delta \vdash o[c, F, L] : (o{:}c)}\ \text{T-NObj}$$

$$\frac{;\, \Delta, n{:}[T] \vdash t : T}{\Delta \vdash n\langle t \rangle : (n{:}[T])}\ \text{T-NThread} \qquad \frac{\Delta' \leq \Delta \quad \Theta \leq \Theta' \quad \Delta \vdash C : \Theta}{\Delta' \vdash C : \Theta'}\ \text{T-Sub}$$

Table 13. Typechecking (1)

26. M. Steffen. *Object-Connectivity and Observability for Class-Based, Object-Oriented Languages*. Habilitation thesis, Technische Faktultät der Christian-Albrechts-Universität zu Kiel, 2006. Submitted 4th. July, accepted 7. February 2007.
27. K. Takeuchi, K. Honda, and M. Kubo. An interaction-based language and its typing system. In C. Halatsis, D. Maritsas, G. Philokyprou, and S. Theodoridis, editors, *Proceedings of PARLE '94*, volume 817 of *Lecture Notes in Computer Science*, pages 398–413. Springer-Verlag, 1994.

A Appendix

A.1 Type checking

Type checking is split into two levels, one on the level of components (cf. Table 13) and one on the level of thread, expressions, and their sub-phrases (cf. Table 14). For components, the rules formalize a judgement of the form $\Delta \vdash C : \Theta$, where Δ is the assumption context and Θ the commintment context. Both associate (class, thread, and object) names with their respective types, where the assumption context takes care of those names which are part of the environmemt, whereas dually Θ is reponsible for the names of the componenent. At the level of threads and expressions, the judgments are of the form $\Delta; \Gamma \vdash t : T$, where Δ is the (assumption) name context, and Γ contains the bindings of the local variables.

A.2 Structural congruence

Components are considered up-to a standard structural congruence, which is formalized in Table A.2. The rule for scope extrusion on the left-bottom is applied under the side-condition that n does not occur free in C_1. The congruence relation is imported into thye reduction relation via the rules of Table 16.

A.3 Traces

A trace of a well-typed component is a sequence of external steps; we write $\Xi_1 \vdash C_1 \stackrel{t}{\Longrightarrow} \Xi_2 \vdash C_2$ when the component $\Xi_1 \vdash C_1$ evolves to $\Xi_2 \vdash C_2$

by executing the trace t. The corresponding rules are given in Table 17. For $\Xi_1 \vdash C_1 \stackrel{\epsilon}{\Longrightarrow} \Xi_2 \vdash C_2$, we write shorter $\Xi_1 \vdash C_1 \Longrightarrow \Xi_2 \vdash C_2$, where ϵ denote the empty trace.

$$\frac{\Gamma;\Delta \vdash c : [\![l_1{:}U_1, \ldots, l_k{:}U_k]\!] \quad \Gamma;\Delta \vdash m_i : U_i \quad m_i = \varsigma(s_i{:}c).\lambda(\vec{x}_i{:}\vec{T}_i).t_i}{\Gamma;\Delta \vdash [\![l_1 = m_1, \ldots, l_k = m_k]\!] : c}\ \text{T-Class}$$

$$\frac{\Gamma;\Delta \vdash c : [\![l_1{:}U_1, \ldots, l_k{:}U_k]\!] \qquad \Gamma;\Delta \vdash f_i : U_i \qquad f_i = \varsigma(s_i{:}c).\lambda().v_\perp}{\Gamma;\Delta \vdash [l_1 = f_1, \ldots, l_k = f_k] : c}\ \text{T-Obj}$$

$$\frac{\Gamma, \vec{x}{:}\vec{T};\Delta, s{:}c \vdash t : T' :: \acute{\Gamma};\acute{\Delta} \quad \Gamma;\Delta \vdash c : T \quad T = [\![\ldots, l{:}\vec{T} \to T', \ldots]\!]}{\Gamma;\Delta \vdash \varsigma(s{:}c).\lambda(\vec{x}{:}\vec{T}).t : T.l}\ \text{T-Memb}$$

$$\frac{\Gamma;\Delta, s{:}c \vdash c : [\![\ldots, l : \mathsf{Unit} \to c', \ldots]\!]}{\Gamma;\Delta \vdash \varsigma(s{:}c).\lambda().\perp_{c'} : c'}\ \text{T-Undef}$$

$$\frac{\Gamma;\Delta \vdash v : c \quad \Gamma;\Delta \vdash c : T \quad \Gamma;\Delta \vdash v' : T.l}{\Gamma;\Delta \vdash v.l := v' : c}\ \text{T-FUpdate} \qquad \frac{\Gamma;\Delta \vdash c : [\![T]\!]}{\Gamma;\Delta \vdash \mathsf{new}\, c : c}\ \text{T-NewC}$$

$$\frac{\Gamma;\Delta \vdash e : T_1 \qquad \Gamma, x{:}T_1;\Delta \vdash t : T_2}{\Gamma;\Delta \vdash \mathsf{let}\, x{:}T_1 = e\, \mathsf{in}\, t : T_2}\ \text{T-Let}$$

$$\frac{\Gamma;\Delta \vdash v_1 : T_1 \quad \Gamma;\Delta \vdash v_2 : T_1 \quad \Gamma;\Delta \vdash e_1 : T_2 \quad \Gamma;\Delta \vdash e_2 : T_2}{\Gamma;\Delta \vdash \mathsf{if}\, v_1 = v_2\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2 : T_2}\ \text{T-Cond}$$

$$\frac{\Gamma;\Delta \vdash v : c \quad \Gamma;\Delta \vdash c : [\![\ldots, l{:}\mathsf{Unit} \to T, \ldots]\!] \quad \Gamma;\Delta \vdash e_1 : T_2 \quad \Gamma;\Delta \vdash e_2 : T_2}{\Gamma;\Delta \vdash \mathsf{if}\, \mathsf{undef}(v.l())\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2 : T_2}\ \text{T-Cond}_\perp$$

$$\frac{}{\Gamma;\Delta \vdash \mathsf{stop} : T}\ \text{T-Stop} \qquad \frac{}{\Gamma;\Delta \vdash () : \mathsf{Unit}}\ \text{T-Unit}$$

$$\frac{\Gamma;\Delta \vdash v : c \quad \Gamma;\Delta \vdash c : [\![\ldots, l{:}\vec{T} \to T, \ldots]\!] \quad \Gamma;\Delta \vdash \vec{v} : \vec{T}}{\Gamma;\Delta \vdash v@l(\vec{v}) : [T]}\ \text{T-CallA}$$

$$\frac{\Gamma;\Delta \vdash n : [T] \qquad \Gamma;\Delta \vdash o{:}c}{\Gamma;\Delta \vdash \mathsf{claim}@(n, o) : T}\ \text{T-Claim} \qquad \frac{\Gamma;\Delta \vdash n : [T]}{\Gamma;\Delta \vdash \mathsf{get}@n : T}\ \text{T-Get}$$

$$\frac{\Gamma(x) = T}{\Gamma;\Delta \vdash x : T}\ \text{T-Var} \qquad \frac{\Delta(x) = T}{\Gamma;\Delta \vdash n : T}\ \text{T-Name}$$

$$\frac{\Delta \vdash o : c}{\Gamma;\Delta \vdash \mathsf{suspend}(o) : \mathsf{Unit}}\ \text{T-Suspend} \qquad \frac{\Delta \vdash o : c}{\Gamma;\Delta \vdash \mathsf{grab}(o) : \mathsf{Unit}}\ \text{T-Grab}$$

$$\frac{\Delta \vdash o : c}{\Gamma;\Delta \vdash \mathsf{release}(o) : \mathsf{Unit}}\ \text{T-Release}$$

Table 14. Typechecking (2)

$$\mathbf{0} \parallel C \equiv C \qquad C_1 \parallel C_2 \equiv C_2 \parallel C_1 \qquad (C_1 \parallel C_2) \parallel C_3 \equiv C_1 \parallel (C_2 \parallel C_3)$$
$$C_1 \parallel \nu(n{:}T).C_2 \equiv \nu(n{:}T).(C_1 \parallel C_2) \qquad \nu(n_1{:}T_1).\nu(n_2{:}T_2).C \equiv \nu(n_2{:}T_2).\nu(n_1{:}T_1).C$$

Table 15. Structural congruence

$$\frac{C \equiv \rightsquigarrow \equiv C'}{C \rightsquigarrow C'} \qquad \frac{C \rightsquigarrow C'}{C \parallel C'' \rightsquigarrow C' \parallel C''} \qquad \frac{C \rightsquigarrow C'}{\nu(n{:}T).C \rightsquigarrow \nu(n{:}T).C'}$$

$$\frac{C \equiv \xrightarrow{\tau} \equiv C'}{C \xrightarrow{\tau} C'} \qquad \frac{C \xrightarrow{\tau} C'}{C \parallel C'' \xrightarrow{\tau} C' \parallel C''} \qquad \frac{C \xrightarrow{\tau} C'}{\nu(n{:}T).C \xrightarrow{\tau} \nu(n{:}T).C'}$$

Table 16. Reduction modulo congruence

$$\frac{C_1 \Longrightarrow C_2}{\Xi_1 \vdash C_1 \overset{\epsilon}{\Longrightarrow} \Xi_2 \vdash C_2}\ \textsc{Internal} \qquad \frac{\Xi_1 \vdash C_1 \xrightarrow{a} \Xi_2 \vdash C_2}{\Xi_1 \vdash C_1 \overset{a}{\Longrightarrow} \Xi_2 \vdash C_2}\ \textsc{Base}$$

$$\frac{\Xi_1 \vdash C_1 \overset{t_1}{\Longrightarrow} \Xi_2 \vdash C_2 \qquad \Xi_2 \vdash C_2 \overset{t_2}{\Longrightarrow} \Xi_3 \vdash C_3}{\Xi_1 \vdash C_1 \overset{t_1 t_2}{\Longrightarrow} \Xi_3 \vdash C_3}\ \textsc{Conc}$$

Table 17. Traces