Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 253 (2009) 127–143
www.elsevier.com/locate/entcs

Wrap Your Objects Safely [1]

Olaf Owe and Gerardo Schneider[2,3]

Department of Informatics
University of Oslo, Norway

Abstract

Despite the effort of researchers on distributed systems, programming languages, and security, there is still no good solution offering basic constructs for guaranteeing minimal security at the programming language level. In particular, the notion of a *wrapper* around an object or component controlling its interaction with the environment has not properly been addressed. This kind of "local firewall" may play two different roles: (1) The untrusted part is what is inside the wrapper; (2) The untrusted part is the environment. In this paper we propose the addition of a language primitive for creating wrapped objects and components, and sketch a formalization based on a minimal object-oriented language for distributed systems using asynchronous communication.

Keywords: Wrappers, security, objects, components

1 Introduction

Open distributed systems and embedded devices offer many new challenges to the research community, from design to correct functioning, including modeling, practical implementation issues, and most notably security. The latter in particular is a crucial aspect to be taken into account, as the uploading or downloading of an application to a device may compromise data privacy, confidentiality or integrity. The above justifies the need to find new solutions to guarantee a minimum of security at run-time. This is far from easy since the *openness* of modern systems implies that we do not know with what possible environments the software will be interacting.

One way to enforce security in open distributed systems, and in Internet in particular, is to use the *sandbox* model of Java. The Java sandbox consists of a set of rules to limit an untrusted applet to execute certain operations when arriving to the site where the browser who called the applet resides. An alternative model to the sandbox is to run downloaded "signed" code only, which means that it is up to

[1] Partially supported by the Nordunet3 project COSoDIS and the EU project Credo.
[2] Email: olaf@ifi.uio.no
[3] Email: gerardo@ifi.uio.no

doi:10.1016/j.entcs.2009.09.032

the user to allow only completely trusted code. Similar models are used in trusted devices like Java smart cards.

The use of such security measures is, however, left to a late state of development, and is independent of the particular application. The use of a sandbox model is in general considered a good programming practice, since it relies on the *separation of concerns* approach. Along the same lines, and to complement the above, we propose a programming language primitive for wrapping objects and components.

We suggest that a *wrapper* consists of a kind of membrane defined "around" an object (or component) in order to isolate it from its environment. Any communication between the *inside* and the *outside* of the membrane is controlled by the operational part of the wrapper, which *filters* (modifies, deletes, adds) messages according to the underlying security policy specified by the wrapper. We can see a wrapped object in two ways: (1) The untrusted part is the object inside the wrapper, so we need to protect the environment from the wrapped object; (2) The untrusted part is the environment, so we wrap the object to protect it from the environment it will run in.

We propose the definition of a language primitive, `safeNew`, that will be responsible for instantiating a wrapped object. The `safeNew` construct is parametrized with the class that the object is an instance of, and with the "controller" of the wrapper. This second parameter of `safeNew` is a special kind of finite automaton that specifies which actions are to be taken for any input/output communication event, or more complex history traces. We call it the *wrapper automaton*. In order to be of practical use, a library with predefined wrappers should be provided to the programmer, together with a language for creating new wrapper automata.

It has long been identified that the shared-state concurrency model is not ideal for open distributed systems. This is despite the fact that one of the most widespread languages used nowadays, namely Java, relies on such model. The natural style for modeling distributed systems is by message-passing and concurrency, for instance objects running concurrently communicating asynchronously through method calls, as in the Creol model [10]. In order to formalize and experiment with our proposal, we have chosen Creol as the underlying language, due to its communication model, its simple semantics, and simulation tools directly related to the operational semantics. In addition, Creol supports dynamic upgrade of classes, allowing new code to be communicated, where wrappers may also protect against unsafe code.

Creol is a modeling and programming language, based on an asynchronous communication model for active, distributed objects with conditional release points and high-level process control. Non-blocking method calls provide efficiency in a distributed setting. The language addresses many of the objections of current object-oriented languages, for instance the inheritance anomaly, the problem of blocked processors waiting for a synchronous call, as well as problematic verification issues [11]. Furthermore, Creol supports dynamic reconfiguration by means of class upgrades. An upgrade may consist of added attributes, added interfaces, added definitions of new methods, as well as redefinitions of old methods. A class upgrade is made by sending a special message into the configuration, which is then spread to

$$
\begin{array}{lll}
CL & ::= & \textbf{class}\ Id\ \textbf{begin}\ Vdecl^{?}\ \{\{\textbf{with}\ Id\}^{?}\ Mtds\}^{*}\ \textbf{end} \\
Type & ::= & Id\{[\{Type\}^{+}_{,}]\}^{?} \\
Vdecl & ::= & \textbf{var}\ \{\{v\}^{+}_{,} : Type\,\{= e\}^{?}\}^{+}_{,} \\
Mtds & ::= & \{Msig == \{Vdecl;\}^{?}\ \bar{s}\}^{+} \\
Msig & ::= & \textbf{op}\ Id\ \{(\{\textbf{in}\ Par\}^{?}\ \{\textbf{out}\ Par\}^{?})\}^{?} \\
Par & ::= & \{\{v\}^{+}_{,} : Type\}^{+}_{,}
\end{array}
$$

Figure 1. BNF for class declarations.

the classes and objects in a distributed manner, and installed independently at the affected objects, without stopping execution. Since the upgrade of classes is done through messages, it means that insertion of new upgrades in a hostile environment may be unsafe, with the result that unsafe code may be installed and executed. By focusing on controlling the flow of messages in and out of wrappers, we then address the issues of unsafe data and code flow.

The main contributions of our paper are: (1) The proposal of a programming language primitive, called `safeNew` to create wrapped objects; (2) An operational semantics of the language primitive and related language constructs, adapting earlier work on Creol to the current setting; (3) The extension of Creol with a novel notion of self-contained component, to accommodate wrapped components.

The paper is organized as follows. Section 2 recalls the relevant parts of Creol, and Section 3 presents a modified operational semantics of the language suitable for our purposes. Section 4 shows how to extend Creol with a primitive for creating wrapped objects and we discuss how to wrap components. We then present an example in Creol in Section 5, where we "protect" what is inside the wrapper from external accessing. Section 6 discusses related work, and the last section concludes.

2 The Creol Language: Syntax

In Creol [11,12,8], objects execute concurrently, encapsulating an execution thread and an internal process queue. Active behavior is interleaved with passive behavior by means of release points. Consequently, elements of basic data types are not considered as objects in the language, and the language includes an underlying functional language for defining data types and associated functions. Objects have identities, which are unique; communication takes place between named objects, and object identities (references) may be exchanged between objects. Creol object variables are typed by interfaces extended with semantic requirements and mechanisms for type control in dynamically reconfigurable systems. The language is strongly typed, ensuring that invoked methods are supported by the called object (when not nil), and that formal and actual parameters match [12]. Objects are dynamically created instances of classes, and multiple inheritance is supported, with late binding of method calls.

The basic syntax of the language is presented in Fig. 1, where we show the BNF grammar for simple class declarations. Curly brackets are used as meta-

Syntactic categories.	*Definitions.*
t in Label	$g ::= wait \mid b \mid t? \mid g \wedge g$
g in Guard	$p ::= x.m \mid m$
p in MtdCall	$\overline{s} ::= \varepsilon \mid s; \overline{s}$
s in Stm	$s ::= (\overline{s})$
v in Var	$\mid \overline{v} := \overline{e} \mid v :=$ **new** $Id(\overline{e})$
e in Expr	$\mid$ **if** b **then** $\overline{s}$ **else** $\overline{s}$ **fi**
m in Mtd	$\mid$ **while** b **do** $\overline{s}$ **od**
x in ObjExpr	$\mid !p(\overline{e}) \mid t!p(\overline{e}) \mid t?(\overline{v}) \mid p(\overline{e}; \overline{v})$
b in BoolExpr	$\mid$ **await** $g \mid$ **await** $p(\overline{e}; \overline{v})$

Figure 2. Syntax for imperative statements.

parentheses, superscript $^?$ for optional parts, superscript * for repetition zero or more times, whereas $\{\ldots\}^+_,$ denotes repetition one or more times with "," as delimiter. Identifiers Id denote interface, class, type, or method names. In Fig. 2 we present the language syntax for imperative statement lists, with typical terms for each category. Overlined terms such as $\overline{s}$, $\overline{e}$, and $\overline{v}$, denote lists of statements, expressions, and variables, respectively. We here ignore class inheritance and interfaces, which are not central to the discussions of this paper. Classes define a number of attributes (with initial values) and a number of method definitions. A method is defined by an imperative statement list, accessing the class attributes, the local variables as well as the input and output parameters of the method (given by the keywords **in** and **out**). In order to allow type correct call-backs, a method may use the implicit *caller* parameter, letting the (minimal) type of the caller be given by the so-called *cointerface* of the method (given by a **with** clause). Parameters, as well as *this*, used for self reference, are read-only.

Assignment, if- and while-constructs follow traditional syntax. Multiple assignments are allowed (as in $x, y := y, x$, which may be used for swapping). Object creation has the syntax $v :=$ **new** $C(\overline{e})$ where C is the class name and $\overline{e}$ the list of actual class parameters, if any. Methods calls may be local, as in $m(...)$, or remote, as in $x.m(...)$ where x is an object expression. Note that remote attribute access is not permitted, as (asynchronous) method interaction is the only communication mechanism of the language.

An asynchronous method call has the syntax $!p(\overline{e})$, if no return values are needed, and $t!p(\overline{e})$ otherwise. In the latter case, the tag t will be assigned a unique tag value identifying the call (relative to the current object). With the guard **await** $t?$ one may test if a return from the callee has arrived, otherwise the guard is false and the current process is suspended and put on the process queue of the current object. Once a return has arrived, the return values can be picked up by the *reply statement*

$t?(\overline{v})$ and assigned to $\overline{v}$. The static typing checks that the variable list $\overline{v}$ has the correct length and type.

The *non-blocking* method call **await** $p(\overline{e};\overline{v})$ (where p is of the form $x.m$ or m) is a short form of $t!x.m(\overline{e})$; **await** $t?$; $t?(\overline{v})$, where t is a fresh (or unused) label. Furthermore, the synchronous and blocking method call $p(\overline{e};\overline{v})$ is a short form of $t!x.m(\overline{e})$; $t?(\overline{v})$, where the unguarded reply statement corresponds to active waiting. Thus synchronous calls can be reduced to asynchronous calls. A local call $m(\overline{e};\overline{v})$ is understood as a remote call to self, $this.m(\overline{e};\overline{v})$ (and measures are taken to provide self reentry). Remote method calls are typically programmed by non-blocking calls, thereby avoiding deadlock and active waiting.

Guards are used to control processor release and may consist of Boolean conditions, return tests, and definite release (*wait*). When a process is released or completed, an enabled process (if any) is chosen from the local process queue. Therefore explicit signaling is not part of the language. The run method of an object is called upon creation, and initiates active behavior. Release points in the run method, for instance an asynchronous recursive call, allow processes in the process queue to be handled. Notice that a method call creates an invocation event message sent to the callee, which results in an activation of the called method, which is placed on the process queue of the callee. And the completion of a method activation generates a return event message sent back to the caller (with the caller and tag value as implicit parameters).

3 The Creol Language: Semantics

This section will present a simplified and adapted version of the original Creol's operational semantics, better suited to our purpose of extending the language (in the next section) with a wrapping primitive and notions of locality and self-contained components. Creol's operational semantics is formally defined as a set of rewriting logic (RL) equations and rules in Maude [6]. RL captures concurrency naturally: given a set of rewriting rules and equations, the rules are non-deterministically applied modulo the equations. If several rules can be applied to distinct subterms, they can be executed in a *concurrent rewrite step*. As a result, concurrency is implicit in RL semantics. Rules and equations may be unconditional or have a conditional on its application. In the latter case the format is $lhs \longrightarrow rhs$ **if** *condition* for rules and $lhs = rhs$ **if** *condition* for equations.

In particular, RL includes a model of object-orientation: objects are represented by terms of the type $\langle o : C \mid a_1 : v_1, \ldots, a_n : v_n \rangle$ where o is the object's identifier, C is its class, the a_i's are the names of the object's fields, and the v_i's are the corresponding values. A system state can then be modeled as a *configuration*, which is defined as a multiset of RL objects and messages representing object communication. It is customary to introduce the configuration sort as a supersort of the object and message sorts, with two constructors ([ctor]):

op none : → Config [**ctor**] .
op _ _ : Config Config → Config [**ctor assoc comm identity**: none] .

Here white-space is used as a convenient infix symbol for the composition operator, and we exploit Maude's built in understanding of associativity, commutativity, and identity (with *none* as identity element).

The operational semantics of Creol is based on this model, representing Creol classes and objects as RL objects, and using messages to reflect method invocations and method returns. It provides an executable simulator for Creol, consisting of the operational semantics rules, corresponding to the imperative statements of the language, as well as run-time system functions such as method binding and message transport.

A Creol object is represented by a Maude object of the form:

$$<\text{o} : \text{C} \mid \text{Att: A, Lvar: L, Pr: S, PrQ: W, InQ: Q, Icnt: I, Ocnt: N} >$$

where o is the object identity, C is the class name, A the state of the attributes (a map from variable names to values), L the state of the local variables, S the statements of the active process, W the process queue, Q the queue of incoming messages, I a system variable used to generate unique tag values for method invocations, and N a system counter used for generating unique identity of new objects (generated by this object). In contrast to the object-oriented model of Maude, classes are represented explicitly, in order to allow explicit control of method binding and inheritance. A Creol class is represented by a Maude object of the form:

$$<\text{C} : \text{Cl} \mid \text{Mtd: M, Att: A} >$$

where C is the class name, M a multiset of method declarations (each with code and local variables), and A a list of attributes (with initial values). For simplicity, we ignore inheritance in this paper, otherwise there would be a field $Inh : S$ defining the inheritance list. In the rules we omit fields not relevant for the rule, following the convention of Full Maude. For instance, the following rule defines object creation in Creol (ignoring class parameters for simplicity):

$$\begin{array}{ll}
(\textsc{New}): & <\text{C} : \text{Cl} \mid \text{Mtd: M, Att: A} > \\
& <\text{O} : \text{C'} \mid \text{Pr: v:=new C; S, Ocnt: N} > \\
\longrightarrow & <\text{C} : \text{Cl} \mid \text{Mtd: M, Att: A} > \\
& <\text{O} : \text{C'} \mid \text{Pr: v:=ob(O,N); S, Ocnt: N+1} > \\
& <\text{ob(O,N)}: \text{C} \mid \text{Att: A+(}\mathit{this} \mapsto \text{ob(O,N))}, \text{Lvar: } \varepsilon, \text{ Pr: run(), PrQ: } \varepsilon, \text{ InQ: } \varepsilon, \\
& \qquad\qquad \text{Icnt: 1, Ocnt: 1} > .
\end{array}$$

where ε denotes an empty list. The right hand side introduces a new object, with identity given by $ob(O, N)$ ("the Nth object generated by O") where ob is a constructor for generating identities. The parent object assigns this identity to the variable v, increases its object counter, and afterwards continues executing the rest of the statements S of its active process. (By our convention, the other attributes of the parent object remains unchanged.) The new object is given default values to all attributes, including the implicit variable *this*, and is started by a synchronous call to its run method. In presence of inheritance, the calculation of attributes A of the new object would be more involved.

(Assign): $\langle O : C \mid \text{Att: } A, \text{ Lvar: } L, \text{ Pr:}(X{:=}E;\ S)\rangle$
$\longrightarrow$ **if** X **in** A **then** $\langle O : C \mid \text{Att: } (A + (X \mapsto \text{eval}(E,\ A + L))), \text{ Lvar: } L, \text{ Pr: } S \rangle$
else $\langle O : C \mid \text{Att: } A, \text{ Lvar: } L + (X \mapsto \text{eval}(E,\ A + L))), \text{ Pr: } S\rangle$ **fi** .

(call): $\langle O : C \mid \text{Att: } A, \text{ Lvar: } L, \text{ Pr: } (T!OE.\text{m}(E);\ S), \text{ Icnt: } I \rangle$
$\longrightarrow$ $\langle O : C \mid \text{Att: } A, \text{ Lvar: } L, \text{ Pr: } (T :=I;\ S), \text{ Icnt: } I{+}1 \rangle$
invoc m(O, I,eval($E, A + L$) **to** eval($OE, A + L$)).

(Message): $(MSG$ **to** $O)\ \langle O : C \mid \text{InQ: } Q\rangle$
$\longrightarrow$ $\langle O : C \mid \text{InQ: } (Q; MSG)\rangle$.

(Bind): $\langle C : Cl \mid \text{Mtd: } M\rangle\ \langle O : C \mid \text{PrQ: } W, \text{ InQ: (invoc m}(E); Q) \rangle$
$=$ $\langle C : Cl \mid \text{Mtd: } M\rangle\ \langle O : C \mid \text{PrQ: } (W; \text{bind(m,}E\text{,}M\text{))}, \text{ InQ: } Q\rangle$
if m **in** M .

(Return): $\langle O : C \mid \text{Att: } A, \text{ Lvar: } L, \text{ Pr: return}(E) \rangle$
$\longrightarrow$ $\langle O : C \mid \text{Att: } A, \text{ Lvar: } \varepsilon, \text{ Pr: } \varepsilon \rangle$
comp (eval((tag,E), $A + L$) **to** L[caller]).

(Reply): $\langle O : C \mid \text{Att: } A, \text{ Lvar: } L, \text{ Pr: } (T?(X);\ S), \text{ InQ: } (Q; \text{comp}(I, E); Q') \rangle$
$\longrightarrow$ $\langle O : C \mid \text{Att: } A, \text{ Lvar: } L, \text{ Pr: } (X :=E;\ S), \text{ InQ: } (Q; Q') \rangle$
if $I{=}{=}\text{eval}(T, A + L)$.

(Guard): $\langle O : \text{C} \mid \text{Att: } A, \text{ Lvar: } L, \text{ Pr: (await G; } S), \text{ PrQ: } W, \text{ InQ: } Q \rangle$
$\longrightarrow$ $\langle O : \text{C} \mid \text{Att: } A, \text{ Lvar: } L, \text{ Pr: } S, \text{ PrQ: } W, \text{ InQ: } Q \rangle$
if enabled(G, $(A + L)$, Q).

(Suspend): $\langle O : \text{C} \mid \text{Att: } A, \text{ Lvar: } L, \text{ Pr: } S, \text{ PrQ: } W, \text{ InQ: } Q \rangle$
$\longrightarrow$ $\langle O : \text{C} \mid \text{Att: } A, \text{ Lvar: } \varepsilon, \text{ Pr: } \varepsilon, \text{ PrQ: } W; (L,\ S), \text{ InQ: } Q \rangle$
if not enabled(S, $(A + L)$, Q).

(Activate): $\langle O : \text{C} \mid \text{Att: } A, \text{ Lvar: } \varepsilon, \text{ Pr: } \varepsilon, \text{ PrQ: } (L,\ S);\ W, \text{ InQ: } Q \rangle$
$\longrightarrow$ $\langle O : \text{C} \mid \text{Att: } A, \text{ Lvar: } L, \text{ Pr: } S, \text{ PrQ: } W, \text{ InQ: } Q \rangle$
if enabled(S, $(A + L)$, Q).

Figure 3. Rules concerning standard constructs and communication. Maude variables are capitalized.

Fig. 3 presents the operational semantics of our version of Creol, apart from object creation and `safeNew`, which are discussed separately. Matching is modulo associativity, commutativity, and identity (ACI) for the multiset constructor, and modulo associativity and identity for the list constructor (with ε as identity element, and semicolon as composition operator). In particular, matching modulo the identity $s;\varepsilon = s$ ensures that left hand sides with pattern $s; S$ (in Pr) match s. For a state L, the notation $L[X]$ denotes the value bound to the variable X, and $+$ is used for state composition (and overriding). Evaluation of terms is done by the

eval function, while *enabled* tests if a guard, statement or statement list is enabled, i.e. it is unguarded or starts with a guard that is satisfied. Programs are assumed to be type correct. Class inheritance and class parameters are ignored, since these concepts are not the focus here. The obvious rules for multiple assignment, if-, and while-statements are omitted. Furthermore, we ignore rules for self-reentry (in the case of local synchronous calls). The operational semantics gives an executable interpreter for Creol programs (when adding Maude definitions of auxiliary functions). The presented semantics is based on code copying; a more space-efficient run-time system could be made by using instruction pointers and code sharing.

A method call results in an invocation message placed in the configuration, and the tag variable is assigned the value of the invocation counter (CALL). The invocation message is transported through the network by a non-deterministic rule (MESSAGE), allowing message overtaking. Method binding is then done by placing a copy of the method activation on the process queue (BIND). The *bind* function will extract a method activation consisting of a copy of the local variable (with initial values) and the code, inserting the statement $return(x)$ at the end where x is the list of formal out parameters; and the actual parameters values are assigned to the formal parameters, including the implicit parameters *caller* and *tag*. The execution of a *return* statement (the last statement in a method) causes a completion message, and the (MESSAGE) rule will ensure transportation to the in-queue of the caller object, which may wait passively for a reply in a suspended process (by means of an await statement) or actively by a reply statement. Enabled processes become activated or reactivated when the active process is suspended/terminated.

4 Syntactic Extensions to Creol

Suitability of Creol

The model of objects running concurrently, communicating asynchronously through methods calls and with processor release points, as presented so far, is too abstract when it comes to modeling certain aspects of the Internet: The Internet involves security issues which escape from the scope of such an abstract model. The first limitation is the fact that all the messages are "thrown" into the system configuration –modeling the network– which are then redirected to the incoming queues of the target objects (allowing overtaking). This is of course reasonable as an abstraction, but not if the aim is to prove, for instance, that a secret message does not end up in the wrong queue. A malicious object may change messages, other objects and even the definition of classes. Moreover, concerning security, as a modeling language Creol also needs to be extended with a model of the attacker, i.e. a process which may add, modify and delete messages and modify the definition of methods and attributes in classes.

First, we make the system fully open (i.e. allowing intrusion mechanisms), by adding the following rules schemas:

(MESSAGE-DELETION): m **to** o ⟶none .

(MESSAGE-CREATION): none ⟶m **to** o .

(MESSAGE-ALTERATION): m **to** o ⟶m' **to** o' .

assuming messages have the form *m to o* where *m* is the message content (either an invocation or a completion with actual parameter values, or an upgrade message) and *o* is the destination of the message. Note that the two last rules are not executable in Maude when *m* and *o* are variables, since the right hand sides are not fully determined. Also notice that since the class upgrade mechanism of Creol is done through message passing, it means that safety of data, as well as of code, is affected by alteration or insertion of messages.

Second, we need to add the notion of locality. Currently all messages go to a global pool, also containing all objects and classes, and there is no way to group them into units with a local pool. Locality is needed, among other things, to model local networks and computers connected to the network. In Maude we can create localities in a very similar way as we create wrappers.

Finally, we need to define a suitable language for defining wrapper automata. This can be done by using the underlying functional data type language of Creol.

Addition of `safeNew`*, wrappers and components*

We show now how to define a notion of *self-contained component*, which may be seen as local subsystem containing all class code needed. We then show how to implement *wrappers* and the language primitive `safeNew`.

We introduce a sort *Component* with the constructor:

op _+_ : Classes System →Component [**ctor**].

where sort *System* is like *Config* but without class declarations, and where *Classes* is a multiset of class declarations. We define sort *Wrapper* as a subsort of *System*:

sort Wrapper .
subsorts Wrapper < System .
op {_|_} : Component Automaton →Wrapper [**ctor**].

Notice that this definition allows nesting of wrappers, in the sense that the inside configuration of a wrapper may again contain inner wrappers. We syntactically extend the Creol language shown in Fig. 2 with the following command: `safeNew` $C(E; FA)$ where E is a list of parameters, as for unwrapped objects, and FA is the wrapper automaton (left underspecified here). We will sometimes omit the parameters E in what follows since their treatment is as for the normal new operation for creating unwrapped objects. At run-time a wrapper is created by a `safeNew` command, creating a new object together with its class as well as the given automaton. Since any message in the configuration is accessible to any (malicious) observer, the wrapper must be created with the relevant classes inside, before sending any message to the configuration. We need, however, to keep the classes accessible to any other process outside the wrapper. We achieve this by copying the relevant classes inside the wrapper:

(SAFENEW): CL +<O: C' |Pr: v:=safeNew C(FA); S, Ocnt: N >

$\longrightarrow$ CL + < O: C' | Pr: v:=ob(O,N); S, Ocnt: N+1 >
{ classes (CL,C) +
< ob(O,N): C | Att: A+(*this* $\mapsto$ ob(O,N)), Lvar:ε, Pr: run(), PrQ: ε, InQ: ε,
Icnt: 1, Ocnt: 1 >
| FA } .

where *CL* is of sort *Classes*, *FA* represents an automaton in a given state (here the initial state), and the *classes* function extracts from CL all class declarations (and their superclasses) of class names textually occurring inside the declaration of C. Notice that the wrapper contains a self-contained unit, in the sense that all code needed is found inside the unit. However, in order to communicate with the outside environment, messages must be imported and exported through the wrapper boundaries. A possible wrapper configuration may then look like $\{< C : CL|... > + < o : C|... > \ (m \ to \ o)\,(m' \ to \ o') \,|\, FA\}$. The syntactic distinction between the configuration inside the wrapper from the outside one guarantees a certain safe-by-construction property, since it is not possible to get into the wrapper unless there is a semantic rule "throwing" messages from the configuration to the inside of the wrapper. The correctness-by-construction is then guaranteed semantically by induction over the set of rules. Also various tests ensuring code security and checking may be placed inside the *classes* function.

A wrapped object will then perform as before with standard Creol rules, without being aware that it is wrapped, including processing of method invocations and reply statements. Method binding is also done as in standard Creol since copies of the relevant class declarations are found inside the wrapper (adjusting the (Bind) rule to the syntax for components). However, we must add rules to control the flow of messages from inside the wrapped component to its environment, and vice versa. The rule for importing into the wrapper is as follows:

(Import): MSG **to** O { CL +SS | FA }
$\longrightarrow$ { CL + SS output(FA,import(MSG **to** O)) | state (FA,import(MSG **to** O)) }
if O **in** SS and accept(FA,import(MSG **to** O)).

where MSG denotes a message content, SS a system state, *output* the output function of a state machine for a given input, *state* the state resulting from the step taken by the automaton, and *accept* the accepting condition. An automaton takes two kinds of inputs: messages directed into the wrapper and messages directed out of the wrapper. To make an automaton distinguish between these, we use the two constructors *import* and *export*, letting *import* encode messages directed into the wrapper, and *export* for messages directed out of the wrapper. In our setting an automaton can be defined in the functional sublanguage of Creol defining constructors for each automaton state (including variable substitutions), and by defining the three functions: *output*, *state*, and *accept*, given an automaton state and input. The rule for exporting from the wrapper is as follows:

(Export): { CL +SS (MSG **to** O) | FA }
$\longrightarrow$ output(FA,export(MSG **to** O))

```
class RWController(db: DataBase)
begin
var free: Bool = true,  readers: ObjSet = ∅,  writer: Obj = null
    pr, pw: Nat = 0    // pending calls to db.read and db.write
with RWClient
  op OR() == await free; if writer ≠ null then free := false;
    await (writer = null); free := true fi;
    readers := readers ∪ {caller}
  op CR() == await (caller ∈ readers);
 readers := readers \ {caller}
  op OW() == await free; free := false;
    await (readers = ∅ ∧ pr = 0 ∧ writer = null);
    free := true; writer := caller
  op CW() == await (pw = 0 ∧ writer = caller); writer := null
  op read(in k: Key out x: Data) == await (caller ∈ readers);
    pr := pr + 1; await db.read(k; x); pr := pr − 1
  op write(in k: Key, x: Data) == await (writer = caller);
    pw := pw + 1; await db.write(k,x); pw := pw − 1
end
```

Figure 4. A consistent read/write class.

$$\{\ CL + SS\ \mid\ \text{state}(FA,\text{export}(MSG\ \textbf{to}\ O))\ \}$$
$$\textbf{if}\ \text{not}\ O\ \textbf{in}\ SS\ \text{and}\ \text{accept}(FA,\text{export}(MSG\ \textbf{to}\ O)).$$

Standard object creation can still be done by the previous (New) rule (adjusted to the component syntax). Thereby objects may be added inside a wrapped component, resulting in components that contain many objects and many classes, in such a manner that all code is found within the wrapper. The resulting system may contain several copies of the classes involved, some inside wrappers and some outside. This has the benefit that each wrapped unit may decide which class upgrades it does accept and which it does not accept. As a result the different class copies may evolve differently and exist in different versions of the same class. The rules given handle this. We have so far not defined components with their own identity which may be manipulated in the context of other components. That is, we cannot import third-parties components, delete them, etc. This stronger notion of component could be added to Creol, but it is not presented in this paper.

5 Example: Readers and Writers

We present now an example that will serve the purpose of illustrating the use of `safeNew` in the second sense explained in the introduction (i.e., we try to protect what is inside the wrapper).

Let us consider an application allowing users to read and write a shared database. In Fig. 4 we show a Creol class *RWController* (taken from [8]) with consistency checking, so no reading is possible while writing and a write waits until all the reads have completed before starting. We assume given a shared database *db*, which provides two basic operations *read* and *write*. Through interface specifications, these are assumed to be accessible for *RWController* objects only. Clients will communi-

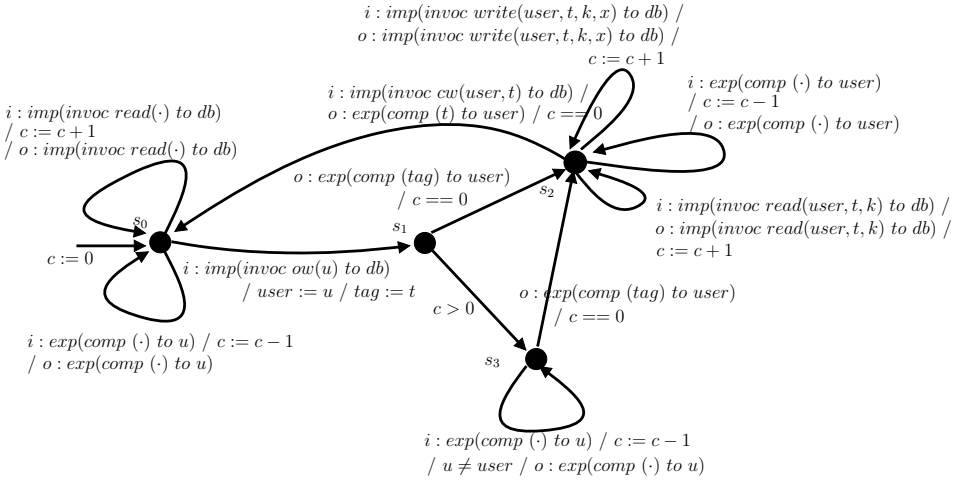

Figure 5. Example: Wrapper automaton.

cate with an *RWController* object to obtain read and write access to the database. *RWController* provides *read* and *write* operations to clients and in addition four methods to synchronize read and write activity: *OR* (OpenRead), *CR* (CloseRead), *OW* (OpenWrite) and *CW* (CloseWrite). A read session happens between invocations of *OR* and *CR* and writing between invocations of *OW* and *CW*. A client is assumed not to terminate unless it has invoked *CR* and *CW* at least as many times as *OR* and *OW*, respectively. To ensure fair competition between readers and writers, invocations of *OR* and *OW* compete on equal terms for a guard *free*. If the condition for reads or writes is unsatisfied, *free* is set to false and the process is suspended.

A program creating an instance of the above class should contain the following line: `rwcons := new RWController(db)`, where `db` is an interface of the `DataBase` class.

We now show how we could obtain the same consistent updating of the database by using wrappers. The program creating the instance of the wrapped object is only slightly changed since it will contain `safeNew` instead of a `new` statement: `rwcons := safeNew DataBase(;Aut)`. Here we create a database object with a wrapper, using the automaton to protect the usage of *read* and *write* operations. The wrapper automaton `Aut`, shown in Fig. 5, takes care of the consistency of the read and write activity. The labels in the edges represent actions taken when the given transition is triggered, and conditions that must be satisfied in order the transition to be taken, separated by "/". We use the notation $i : imp(invoc\ m(u, E)\ to\ o)$ to denote that the automaton reads the invocation to method m from the outside pool of messages in order to import it inside the wrapper, where o is the callee. The message should match $m(u, E)$, where the first parameter u identifies the caller and E are other parameters. Whenever the parameters are not useful for the current state of the automaton, we write $m(\cdot)$. When taking a transition labeled with $o : imp(invoc\ m(u, E)\ to\ o)$, the automaton imports the invocation by u of method $m(E)$, to the inside of the wrapper. Again we write $m(\cdot)$ as before if the param-

eters are not needed. In order to export messages from the wrapper we use *exp*, and again combined with i and o the automaton reads from the inside of the wrapper and writes to the outside. There are then different possibilities, for instance $i : exp(comp\ m(\cdot)\ to\ u)$ reads the method call $m(\cdot)$, from user u, from inside the wrapper, whose execution has been completed, and $o : exp(comp\ m(\cdot)\ to\ u)$ sends it to the outside of the wrapper (directed to user u). Besides, the transitions of the wrapper automaton contains Boolean conditions, in the example $c == 0$ and $c > 0$, where c is a local variable for counting pending reads and writes. Whenever a certain parameter is only used locally (at one particular transition) we use implicit local matching, and when matching is needed among different transitions, we do it explicitly by using extra variables. For instance the transition from s_2 to the initial state s_0 contains labels $i : imp(invoc\ cw(user, t)\ to\ db)$ and $o : exp(comp\ (t)\ to\ user)$. In this case t is a tag used for matching (locally) the invocation of *cw* with its completion, while *user* is here matched with the assignment done in the transition from state s_0 to s_1. We use $(\cdot)$ to denote any pending message in the pool. For instance, $i : exp(comp\ (\cdot)\ to\ u)$, in the loop transition in state s_3, reads any completed method call from user u, from inside the wrapper.

The automaton then allows all the reads to start till a request to start a write ($ow(u)$) is taken from the outside pool of messages. Before throwing the message into the wrapper, the automaton waits until all the pending reads finish and then gives exclusive access to the writer to update the database. The counter c guarantees that the writer only writes when no pending reads remains (when $c == 0$). Once the user getting the exclusive write to the database (in the automaton is represented by *user*) gets access, then he can have an unlimited number of reads and writes in any order till he decides to finish his exclusive access (by sending a message $cw(\cdot)$). The automaton then finishes his access after all his pending reads and writes are completed (taken care by all the transitions from s_2 to s_2).

Notice that in the automaton we have not included any check that only registered readers and writers are allowed to access the database, but this may be easily added. We have shown a picture of the wrapper automaton since it is more readable, but by using the functional language of Creol we can define finite state machines as the input/output automaton used in this example (as done in [9]).

6 Related work

The idea of a programming language primitive to create wrapped objects controlling their interaction with the environment they are in is new, to the best of our knowledge. However, the general idea of wrappers and controlled communication is not new at all, including "boxed" calculi, behavioral interfaces, different ad-hoc filters, and even aspect-oriented programming. In what follows we will briefly discuss a few representative works related to the concept of wrapper. We will also mention a few approaches that may be used as wrapper automata.

Composition-filters [1] enable an aspect-oriented programming technique where different aspects are expressed in filters in a declarative manner. Different aspects

can be expressed as filters, which can then be composed together and attached to the objects programmed in different object-oriented languages. Composition filters are very similar to our wrappers, with the main difference being that `safeNew` is a primitive of a programming language, and composition filters cross-cut the object definition.

Undoubtedly many "boxed" calculi are based on the idea of a membrane, or box, which acts as a wrapper. These include mobile ambients [3], boxed ambients [2] and the Seal calculus [4]. The above calculi concentrate on the construction and communication of ambients, including mobility through ambient migration. This is, however, a feature not needed for modeling web services, which is one of our domain of application. Our setting is based on objects communicating asynchronously through message passing (method calls) and although it is possible to model objects (e.g. by using input channels in the Seal calculus), none of the above calculi are inherently object-oriented. Moreover, in these calculi messages need to be coded as ambients, adding an extra complexity on the design.

The expressive power of aspect-oriented programming (AOP) [13] is different from that of wrappers, as it may involve control of low-level issues such as the number of assignments to particular variables, which we have not addressed with our wrappers. On the other hand, a wrapper may remove or replace a call to or from an environment, for instance in order to make incompatible components compatible. This is not possible with AOP. The notion of wrapper has been motivated by the desire to build abstract protection concepts with a simple semantics which may be exploited by formal program analysis tools. However, the ideas presented may also be reused to build more low-level wrappers, reacting on for instance assignments, but then with a less modular semantics.

We have not given a language for writing wrapper automata since it can be encoded in Maude (and thus in Creol) in the same manner as presented in [9]. Besides, many existing formalisms may be used to specify wrapper automata depending on the intended use, as for instance input/output automata, security automata [19], interface automata [7], and edit automata [14].

Moreover, wrappers may also be seen as contracts, in particular *deontic electronic contracts*. This kind of contract refers to an agreement between parties where it is established what are the obligations, permissions and prohibitions of each party. Contracts may be written in a contract logic, as for instance the ones presented in [5,18], and the wrapper may consist of an automaton automatically extracted from such contract specification. This seems to be particularly useful for wrapping components [17].

Java platforms do already contain a kind of wrapper, as the whole point of these platforms is to protect untrusted code in some sandbox, where the sandbox wraps applets. The main difference with our approach is that we wrap individual objects (and those class instances created inside the wrapper), and that instead of one Security Manager doing the wrapper, we allow a custom wrapper per object.

7 Final Remarks

In this paper we have proposed a new programming language primitive to create wrapped objects. The rationale behind this is that we believe developers must have at their hands the possibility of controlling certain security issues when programming in insecure environment, or when using components issued by third-parties. The main theoretical contribution is the fact that such a primitive may be added to a language in a way that it is correct-by-construction. As a proof of concept we have shown how the `safeNew` primitive may be added to Creol, a highly asynchronous object-oriented language with formal semantics in rewriting logic. For this purpose Creol has been extended with a suitable notion of self-contained component, allowing nested component. The general ideas may be applied to other languages than Creol and other communication models, including event-based languages, synchronous communication, or other component models. The extension to shared variable systems is less trivial and would require a less modular semantics.

An interesting application domain, besides the Internet, is embedded systems. In particular, the state-of-the-art of open smart cards allowing post-issue upload of multiple-applets is unreliable from the security point of view. The main problem is that the current model of firewalls between applets does not work properly, and many attacks are possible due to failures in the model, the implementation of the virtual machine, or other related bugs [15,16,20]. We do not claim that our proposal will solve all those problems, but that at least will mitigate some of them.

The definition of wrappers as we did in Creol could be the starting point of defining different layers of localities. In fact local networks and computers may be represented as wrappers, without the notion of automaton controlling the input and the output. The only issue will be the addition of a local pool where only explicit rules will allow a given message to navigate in or out from the wrapper. We will need to extend it with an identifier, since localities may need to be referenced. Firewalls could be added in a straightforward manner: a locality with a firewall will essentially be the same as a wrapper as we defined it for the `safeNew` (with the automaton) but over a collection of objects and classes. Thus, firewalls are instances of wrappers in the second sense mentioned in the introduction, where the untrusted part is the environment.

A third use of the idea of wrapper is in the context of components. When a component is deployed, its user can only rely on that the component satisfies its specification. As a self-contained executable (and black-box) unit, the user can at least guarantee that the component behaves at run-time according to its specification. One way to do so is to wrap the component when it is received with its specification and let the wrapper warn in case of a bad behavior.

Our decision of using automata-like formalisms for controlling the wrapper is motivated by the fact that in many cases we may use standard verification techniques, like model checking, to prove that the wrapper satisfies certain properties. In the reader and writer example (Section 5), the class definition in Fig. 4 has been shown correct using a Hoare-like reasoning style in [8]. Though we have not per-

formed the exercise, we believe proving the same properties in our wrapper using model checking will be much easier (and fully automatic).

Future work includes: (1) Addition of a library with "standard" wrappers; (2) Development of a full language for writing automata and wrappers; (3) Use of existing security-like automata as wrapper automata; (4) Application to a real case study, including applications from Internet and smart cards.

Acknowledgement

We are indebted to the anonymous referees for helpful comments, and to Pablo Giambiagi for insightful discussions and participation in an early draft of this paper.

References

[1] M. Aksit, K. Wakita, J. Bosch, L. Bergmans, and A. Yonezawa. Abstracting object interactions using composition filters. In *ECOOP Workshop*, volume 791 of *LNCS*, pages 152–184, 1993.

[2] M. Bugliesi, G. Castagna, and S. Crafa. Boxed ambients. In *TACS*, volume 2215 of *LNCS*, pages 38–63. Springer, 2001.

[3] L. Cardelli and A. D. Gordon. Mobile ambients. In *FoSSaCS*, volume 1378 of *LNCS*, pages 140–155. Springer, 1998.

[4] G. Castagna, J. Vitek, and F. Z. Nardelli. The seal calculus. *Inf. Comput.*, 201(1):1–54, 2005.

[5] P. F. Castro and T. S. E. Maibaum. A complete and compact propositional deontic logic. In *ICTAC*, volume 4711 of *LNCS*, pages 109–123. Springer, 2007.

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

[7] L. de Alfaro and T. A. Henzinger. Interface automata. In *ESEC/FSE'01*, pages 109–120. ACM, 2001.

[8] J. Dovland, E. B. Johnsen, and O. Owe. Verification of concurrent objects with asynchronous method calls. In *SwSTE'05*, pages 141–150. IEEE C.S., Feb. 2005.

[9] I. Grabe, A. Torjusen, and M. Steffen. Executable interface specifications for testing asynchronous creol components. Technical Report 375, Dept. of Informatics, Univ. of Oslo, 2008.

[10] E. B. Johnsen, J. C. Blanchette, M. Kyas, and O. Owe. Intra-object versus inter-object: Concurrency and reasoning in creol. In *2nd Intl. Workshop on Harnessing Theories for Tool Support in Software*, ENTCS, 2008. To appear.

[11] E. B. Johnsen and O. Owe. An asynchronous communication model for distributed concurrent objects. *Soft. and Syst. Modeling*, 6(1):35–58, Mar. 2007.

[12] E. B. Johnsen, O. Owe, and I. C. Yu. Creol: A type-safe object-oriented model for distributed concurrent systems. *Theoterical Computer Science*, 365(1–2):23–66, Nov. 2006.

[13] G. Kiczales, J. Irwin, J. Lamping, J.-M. Loingtier, C. V. Lopes, and C. Maeda. Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming*, volume 1241 of *LNCS*, pages 220–242. Springer-Verlag, 1997.

[14] J. Ligatti, L. Bauer, and D. Walker. Edit automata: enforcement mechanisms for run-time security policies. *Int. J. Inf. Sec.*, 4(1-2):2–16, 2005.

[15] M. Montgomery and K. Krishna. Secure object sharing in java card. In *WOST'99*, pages 14–14. USENIX Association, 1999.

[16] W. Mostowski and E. Poll. Malicious code on Java Card smartcards: Attacks and countermeasures. In *CARDIS*, volume 5189 of *LNCS*, pages 1–16. Springer, 2008.

[17] O. Owe, G. Schneider, and M. Steffen. Components, objects, and contracts. In *SAVCBS'07*, ACM Digital Library, pages 91–94, 2007.

[18] C. Prisacariu and G. Schneider. A formal language for electronic contracts. In *FMOODS'07*, volume 4468 of *LNCS*, pages 174–189. Springer, June 2007.

[19] F. B. Schneider. Enforceable security policies. *ACM Trans. Inf. Syst. Secur.*, 3(1):30–50, 2000.

[20] M. Witteman. Java card security. *Information Security Bulletin*, 8:291–298, October 2003.