To appear in TTSS'08

Intra-Object versus Inter-Object: Concurrency and Reasoning in Creol*

Einar Broch Johnsen[a,1], Jasmin Christian Blanchette[b,2], Marcel Kyas[a,3] and Olaf Owe[a,4]

[a] *Department of Informatics, University of Oslo, Norway*

[b] *Institut für Informatik, Technische Universität München, Germany*

Abstract

In thread-based object-oriented languages, synchronous method calls usually provide the mechanism to transfer control from caller to callee, blocking the caller until the call is completed. This model of control flow is well-suited for sequential and tightly coupled systems but may be criticized in the concurrent and distributed setting, not only for unnecessary delays but also for the reasoning complexity of multithreaded programs. Concurrent objects propose an alternative to multithread concurrency for object-oriented languages, in which each object encapsulates a thread of control and communication between objects is asynchronous. Creol is a formally defined modeling language for concurrent objects which clearly separates intra-object scheduling from inter-object communication by means of interface encapsulation, asynchronous method calls, and internal processor release points. This separation of concerns provides a very clean model of concurrency which significantly simplifies reasoning for highly parallel and distributed object-oriented systems. This paper gives an example-driven introduction to these basic features of Creol and discusses how this separation of concerns influences analysis of Creol models.

1 Introduction

Inter-process communication is becoming increasingly important with the rise of distributed computing over the Internet and local area networks. Object-oriented programming is the dominant paradigm for concurrent and distributed systems and has been recommended by the RM-ODP [28], but traditional models of object interaction and concurrency may be criticized in the distributed setting.

Creol is an executable modeling language that aims at uniting object orientation and distribution in a more natural way [30]. It is object-oriented in the sense that classes are the fundamental structuring unit and that all interaction is by means

* This work has been supported by EU-project IST-33826 *CREDO: Modeling and analysis of evolutionary structures for distributed services* (http://credo.cwi.nl).

[1] Email: einarj@fi.uio.no

[2] Email: blanchet@in.tum.de

[3] Email: kyas@fi.uio.no

[4] Email: olaf@fi.uio.no

of method calls between named objects. Moreover, it features multiple inheritance and late binding. What sets Creol apart from other object-oriented languages is its concurrency model: the language combines the concurrent object model of concurrency with interface encapsulation and processor release points. In the concurrent object model, each object executes on its own virtual processor and objects communicate using *asynchronous method calls*. When an object A calls a method m of an object B, it sends an invocation message to B along with arguments. Method m executes on B's processor and sends a reply to A when it is finished, with return values. Object A may continue executing while it is waiting for B's reply. Compared to standard synchronous method calls this approach leads to increased parallelism in a distributed system, where objects may be dispersed geographically. As usual in object-oriented languages, object identities may be passed around between objects.

Creol offers a very flexible model of encapsulation based on interfaces. An interface exposes some of an object's methods to the environment. Fields are not exposed, and may only be accessed by the object itself. However, an object can offer several interfaces, and each interface may require a *cointerface*; i.e., a static requirement on the caller which enables call-back between peer objects in the distributed environment. Thus, asynchronous method calls through interfaces is the only means of *inter-object* communication. Creol's interface and cointerface concepts ensure that method calls and call-backs are type-safe [35].

Internally, a concurrent object contains a set of processes which are executed in some interleaved order. These processes result from method calls to the object, but the order in which they are executed may depend dynamically on delays and instabilities in the environment. At most one process may execute on the object's virtual processor at any given time, and the executing process may release processor control by means of explicitly declared processor release points. Processes may communicate with each other via the object's state; the values of these local fields may also influence the scheduling of processes through guards. This "cooperative" approach to intra-object concurrency has the advantage that while a method is executing, it can assume that no other processes are accessing the object's fields between its release points. This leads to a programming and reasoning style reminiscent of monitors [9,27], but simpler because it abstracts from explicit signaling to other processes. The use of processor release points in concurrent objects also leads to increased parallelism when objects are waiting for replies, and allows objects to easily combine active and reactive behavior in a peer-to-peer manner [31].

This paper presents an overview of Creol's concurrency model and reasoning system. Focus is on the basic *inter-object* communication model of interfaces and asynchronous method calls, and on the *intra-object* synchronization model. We refer the reader to other papers for the discussion of advanced features such as, e.g., class inheritance [35], specification [29], dynamic class updates [33,42] and type checking [35,36]. The paper then discusses how this model supports analysis of Creol models by means of query-driven executable analysis and a compositional proof system.

The rest of this paper is structured as follows: Section 2 briefly compares Creol's concurrency model with traditional approaches to inter-process communication. Section 3 introduces Creol by example. Section 4 presents the language syntax and informally describes its semantics. Section 5 gives an overview of analysis tech-

niques for Creol models. Finally, Section 6 concludes the paper.

2 Interaction and Synchronization

The two main interaction models for distributed processes are remote method invocation (RMI) and message passing [5]. For tightly coupled systems, shared memory models are also an option, but they do not generalize well to distributed environments where communication takes time and is unreliable.

With remote method invocations, the thread of control is transferred with the call and caller activity is blocked until the return values from the call have been received. In this setting, intra-object concurrency can be handled in various ways. In Java [24], where concurrency is achieved through multithreading, methods can be declared to be serialized by using the `synchronized` keyword, which ensures that only one (serialized) method is active in an object at a time. For non-serialized Java methods, the interference problem related to shared variables reemerges when several threads operate simultaneously in the same object. Consequently, reasoning about Java programs is highly complex [2,12]; safety is by convention rather than by language design [10]. Verification considerations therefore suggest that all methods should be serialized, as in Hybrid [39]. However, an object which makes a remote method call must then wait for the call's return before it can proceed with its execution; any other activity in the object is prohibited while waiting. This waiting imposes a severe limitation in the distributed setting; external delays and instabilities may cause unnecessary waiting. Furthermore, this blocking of internal activity makes it difficult to combine active behavior in an object with the processing of requests from the environment, the combination of which is typical for the peer objects one would expect in the distributed setting.

In contrast to remote method invocation, message passing does not transfer control between the communicating parties. A method call can be modeled in this setting by an invocation and a reply message. Message passing may be synchronized, as in Ada's rendezvous mechanism, in which case both the sender and the receiver must be ready before communication can occur. Hence, the objects synchronize on the message transmission. Remote method invocation can be captured in this model if the calling object is blocked between the two synchronized messages representing the call [5]. If the calling object is allowed to proceed for a while before resynchronizing on the reply message, we obtain a model of method calls that resembles future variables [6,41] or eager invocation [19]. For distributed systems, even this form of synchronization may easily result in unnecessary delays through the blocking of internal activity.

Message passing can also be asynchronous. In this case, message emission is always possible, regardless of when the receiver accepts the message. This approach is well-known from the Actor model [26,3]. However, actors do not distinguish replies from invocations, so capturing method calls with actors quickly becomes unwieldy. Languages that support future variables are usually based on asynchronous message passing; the caller's activity is synchronized with the arrival of the reply message rather than with its emission, and the activities of the caller and the callee need not directly synchronize [7,11,16,41]. This approach seems well-suited for distributed

environments, because it provides execution flexibility; the local execution adapts to the network's inherent latency and unreliability.

Creol's asynchronous method invocation mechanism is implemented in terms of asynchronous message passing. Asynchronous method calls resemble asynchronous message passing as they also avoid blocking, but they provide a more structured interaction model. With asynchronous method calls, object activities may be defined and invoked as usual in object orientation, and code may be structured in terms of inheritance and late binding. Furthermore the separation of execution threads and objects, as done in thread-based languages such as Java, compromises the modularity and encapsulation of objects, thereby resulting in a low-level style of programming. This is avoided by encapsulating control within asynchronously communicating concurrent objects, a very natural model for the distributed setting. Creol is a modeling language based on asynchronously communicating concurrent objects which allows processor release at synchronization points. Compared to other concurrent object models, this reduces the waiting for replies inside objects by allowing different activities to be interleaved inside the concurrent object. In fact any method in Creol may be invoked both synchronously and asynchronously. It is the caller's decision when to synchronize, this decision can even be made at runtime.

The language focuses on high-level programming constructs which adapt to the possible delays in an underspecified environment, abstracting from the exact details of local scheduling. Creol's implicit mutual exclusion within an object and its forced hiding of the object's internal fields makes interference-freedom tests unnecessary [5,20]. Reasoning about a Creol program involves establishing interface properties from invariants for the program's classes. Knowledge of another object is purely based on the known interface of that object, so the external point of view is restricted to the type information of each pointer. Due to the concurrency model discussed above, reasoning about concurrent programs in Creol is based on local proof rules. An advantage of this approach is that the proof system is compositional [18]. In Creol, concurrent objects are independent units of composition.

3 Example: Producers and Consumers

Before reviewing Creol's syntax in detail, let us look at a simple example. Consider a buffer which is shared between a producer and a consumer object. The producer writes to the shared buffer and the consumer reads data from the buffer as it is written. Although it does not demonstrate all the features found in Creol, the example presents most of Creol's key concepts. The example consists of three interfaces and four classes. Let us start with the interfaces:

```
interface WritableBuffer
begin
with Producer
  op put(in x: Int)
end
```

```
interface ReadableBuffer
begin
with Consumer
  op get(out x: Int)
end
```

```
interface Buffer
  inherits WritableBuffer, ReadableBuffer
begin end
```

The `WritableBuffer` interface declares the signature of a `put` method that has an input parameter x of type `Int`. The `ReadableBuffer` interface declares the signature of a `get` method that has an output parameter y of type `Int`; i.e., y will be the return value from the call. The `with Producer` and `with Consumer` clauses declare *cointerface requirements*, which express that only objects of type `Producer` may invoke the `put` method and similarly only objects of type `Consumer` may invoke the `get` method. We omit the `Producer` and `Consumer` interfaces, which do not export any methods, and proceed with the `Producer` and `Consumer` classes.

```
class Producer(buf: WritableBuffer) implements Producer
begin
  var i: Int :=1;
  op run == buf.put(i;); i :=i +1; !run()
end

class Consumer(buf: ReadableBuffer) implements Consumer
begin
  var sum: Int :=0;
  op run == var j: Int; buf.get(; j); sum :=sum +j; !run()
end
```

The `Producer` class provides the `Producer` interface and takes an object that supports the `WritableBuffer` interface as parameter. Its `run` method repeatedly calls `put` on the `WritableBuffer` object with arguments 1, 2, 3, ... These calls are synchronous; i.e., the `run` process is blocked until the method has returned. By prefixing these calls by `await`, one would obtain non-blocking calls, as demonstrated further below. Objects in Creol are active; the `run` method is automatically invoked when `Producer` is instantiated. The `Consumer` class resembles `Producer`. Instances of `Consumer` get their data from an object that supports the `ReadableBuffer` interface, one integer at a time, and compute the sum of the data received from the buffer. The semicolon in `buf.get`$(;j)$ indicates that j is an output argument (`get` has no input argument).

```
class OneSlotBuffer implements Buffer
begin
  var value: Int, full: Bool
  op init == full :=false
  with Producer op put(in x: Int) == await ¬full; value, full :=x, true
  with Consumer op get(out x: Int) == await full; x, full :=value, false
end
```

The `OneSlotBuffer` class supports the `WritableBuffer` and `ReadableBuffer` interfaces, and implements the `put` and `get` methods. The methods are annotated with their respective cointerface requirements. The class also declares two fields `value` and `full` and a parameterless `init` method. This is the constructor method which is called before `run` is invoked. The initial `await` statements of `put` and `get` provide enabling conditions. Thus the execution of an invocation of `put` must wait for `full` to be false, and the execution of an invocation of `put` for `full` to be true.

The buffer's role is to synchronize the producer and the consumer, ensuring that the consumer doesn't read data that the producer hasn't generated yet and that the producer doesn't overwrite data that the consumer hasn't read. In this example, the

single slot buffer can store at most one data item at a time — a more realistic (and more efficient) `Buffer` class implementation would typically use a circular buffer internally.

```
class Main
begin
  op run == var buf: Buffer, prod: Producer, cons: Consumer;
    buf :=new OneSlotBuffer; prod :=new Producer(buf);
    cons :=new Consumer(buf)
end
```

To launch the program, we instantiate the `Main` class. This can be achieved in the Creol interpreter using a special command that takes the name of a class and optional arguments corresponding to the class parameters. The `run` method, which is invoked automatically when `Main` is instantiated, starts by creating a `OneSlotBuffer` instance. Object creation connects the types of references to the classes of objects. The method then creates one `Producer` and one `Consumer` instance, both initialized with a reference to the buffer. At that point, the producer's and the consumer's `run` methods are invoked, giving rise to two nonterminating activities that exchange data through the `Buffer` object. Producer–consumer synchronization works as follows: If the consumer calls `get` before the producer calls `put`, then `full` is false and the `get` method invocation is suspended by the **`await`** `full` statement. This enables `put` to execute. When `put` returns, `full` has become true and `get` may resume its execution. Similarly, if the producer calls `put` twice before the consumer calls `get`, the second `put` invocation is suspended by **`await`** `¬full`. The **`await`**-statements explicitly declare processor release points.

Creol's concurrency model makes it easy to combine active and reactive behavior in the same object. The `run` method initiates the object's active behavior, whereas its exported methods are available to the environment. We illustrate this by the following version of the `Consumer` class:

```
class Consumer (buf: ReadableBuffer) implements Consumer
begin
  var sum :Int;
  op init == sum :=0;
  op run == var j: Int; while true do await buf.get(; j); sum :=sum +j end
  with Any op getSum(out s: Int) == s :=sum
end
```

In this version of the class, a method `getSum` returns the sum computed so far, and initialize `sum` by an `init` method. Note that `getSum` has a cointerface `Any`. This is the supertype of all interfaces and does not impose any requirements on the caller, so any object may call `getSum`. In contrast, only `Producer` objects could call the `put` method on `Buffer` objects.

The introduction of the `getSum` method has led to one more change to the `Consumer` class. The synchronous method call `buf.get`$(; j)$ in `run` has been replaced by the statement **`await`** `buf.get`$(; j)$, which invokes `get` asynchronously, releases the processor while it waits for the reply to this call, and then retrieves the return value. Thanks to the explicit processor release, the object can service incoming calls to `getSum` while waiting for `buf`'s reply.

Syntactic categories	*Definitions*
$C, I, m \in \mathsf{Names}$	$IF ::= \texttt{interface}\ I\ [\texttt{inherits}\ \overline{I}]\ \texttt{begin}\ \{\texttt{with}\ I\ \overline{Sg}\}\ \texttt{end}$
$t \in \mathsf{Label}$	$CL ::= \texttt{class}\ C\ [\overline{x:I}]\ [\texttt{inherits}\ \overline{C}\ [(\overline{e})]]\ [\texttt{implements}\ \overline{I}][\texttt{contracts}\ \overline{I}]$
$g \in \mathsf{Guard}$	$\quad \texttt{begin}\ \{\texttt{var}\ \{x:I\ [:=e]\}\}\ \overline{M}\ \{\texttt{with}\ I\ \overline{M}\}\ \texttt{end}$
$p \in \mathsf{MtdCall}$	$M ::= Sg == [\texttt{var}\ \{x:I\ [:=e]\};]\ \overline{s}$
$s \in \mathsf{Stmt}$	$Sg ::= \texttt{op}\ m\ ([\texttt{in}\ \overline{x:I}][\texttt{out}\ \overline{x:I}])$
$x \in \mathsf{Var}$	$g ::= b \mid t? \mid g \wedge g \mid g \vee g$
$e \in \mathsf{Expr}$	$s ::= \texttt{begin}\ \overline{s}\ \texttt{end} \mid \overline{s} \square \overline{s} \mid x := e \mid x := \texttt{new}\ C[(\overline{e})]$
$o \in \mathsf{ObjExpr}$	$\mid \texttt{skip} \mid \texttt{if}\ b\ \texttt{then}\ \overline{s}\ [\texttt{else}\ \overline{s}]\ \texttt{end} \mid \texttt{while}\ b\ \texttt{do}\ \overline{s}\ \texttt{end}$
$b \in \mathsf{BoolExpr}$	$\mid [t]![o.]m(\overline{e}) \mid t?(x) \mid \texttt{release} \mid \texttt{await}\ g \mid [\texttt{await}][o.]m(\overline{e};\overline{x})$

Figure 1. The language syntax. Terms such as $\overline{e}$, $\overline{x}$, and $\overline{s}$, denote lists over terms of the corresponding syntactic categories, $\{\ldots\}$ denotes lists over larger syntactical elements, and $[\ldots]$ denotes optional elements. Elements in a list are separated by a comma (except statement in statement lists by semicolon).

4 The Creol Syntax

This section presents Creol's syntax and informally explains its semantics. References to further presentations of language aspects and to the type system and formal semantics are given in Section 4.5. A Creol program consists of a list of interface and class declarations. Before looking at the syntax given in Figure 1, we introduce a few basic syntactic categories: identifiers and data types. The set Names of *identifiers* are used to name interfaces, classes, methods, fields, parameters, local variables, and method call labels. The set Types comprises the built-in data types **Bool**, **Int**, **Char**, **Float**. In addition, the name of any interface may be used as a type. If $T, \ldots, T' \in \mathsf{Types}$ are types, then $\texttt{List}[T]$ and $\texttt{Set}[T]$ are lists and sets of type T and $T \times \ldots \times T'$ is a tuple of types $T, \ldots, T'$. For modeling intra-object data manipulation, we assume given a language Expr of side effect free expressions, including operations on natural numbers, lists, etc. This language is not discussed in this paper. In the presentation of the language below, some syntactic elements are optional. The optional elements are specified in Figure 1.

4.1 Interface Declarations

The syntax of an *interface declaration* is given by the production *IF* in Figure 1. The *cointerface* k of a method is an interface that the calling object is required to implement; using the implicit **caller** parameter, the callee can then make callbacks to the calling object through interface k [29]. In order to allow any object to call the methods of an interface, the cointerface is the empty interface **Any**, which is implicitly supported by all classes.

A *method signature* Sg is given by $\texttt{op}\ m\ (\texttt{in}\ \overline{x:I}\ \texttt{out}\ \overline{y:J})$, with $m \in \mathsf{Names}$, $\overline{x}, \overline{y} \in \mathsf{Var}$, and $\overline{I}, \overline{J} \in \mathsf{Types}$. Instead of returning a value directly, Creol methods may assign values to one or several output parameters declared using the **out** keyword. If the method has neither input nor output parameters, the parentheses around the empty parameter list may be omitted.

4.2 Class Declarations

The syntax of a *class definition* is given by the production *CL* in Figure 1. A class defines a number of methods which work on a local state. These methods

may include a method `init`, used to initialize objects of the class, and a method `run`, used to define active behavior. If defined, these methods are automatically invoked at creation time; first `init`, then `run`. A class may declare parameters and fields $x : I$. The parameters must be provided at object creation time and serve as arguments to the constructor method `init`, but otherwise they behave like read-only variables. Parameters and fields are not accessible to other objects.

Whereas interfaces only declare method signatures, classes also contain method bodies. The set $\overline{M}$ of method declarations contains elements with the syntax $Sg ==$ `var` $x : T$`:=`e; $\overline{s}$ where Sg is the method signature, $x \in \mathsf{Var}$ is a local variable of type $T \in \mathsf{Types}$, initialized to e, and $\overline{s} \in \overline{\mathsf{Stmt}}$ is a statement list. Methods declared before the first `with` clause can only be called by the object itself. Methods declared in the scope of a clause `with I` require `I` as an interface of the caller.

A class C may inherit fields and methods from one or several superclasses, but methods may be overridden. The interfaces supported by the class are specified in the `implements` and `contracts` clauses. The class must provide implementations for all methods declared by these interfaces, with matching cointerface requirements. Note that in Creol, inheritance and subtyping do not coincide: Classes that inherit from a class C are not required to support the interfaces in the `implements` clause of C. This stands in contrast with the concept of inheritance as found in Java or $C^\sharp$, but allows more flexible code reuse without breaking behavioral requirements. If we want to force a certain interface (and its behavioral specification) upon a class and all its direct and indirect subclasses, the interface is declared in the `contracts` clause instead of the `implements` clause. Thus, the interfaces contracted by a class are inherited by its subclasses but the interfaces implemented by a class are not.

4.3 Basic Statements

The imperative statements in Creol are now briefly introduced. In addition to standard statements, Creol offers a few statements that are only relevant in a concurrent or distributed context. Basic statements are discussed in this section and composition operators in Section 4.4.

Let $\overline{x} \equiv x_1, \ldots, x_n \in \overline{\mathsf{Names}}$ be a list of variables and $\overline{e} \equiv e_1, \ldots, e_n \in \overline{\mathsf{Exp}}$ be a list of expressions such that each e_i is type-compatible with the corresponding x_i. Assignment has the following syntax: $\overline{x} := \overline{e}$. If $n > 1$, the assignments to $x_1, \ldots, x_n$ are performed simultaneously. Thus, the statement $a, b := b, a$ swaps the values of a and b. Since expressions do not have side effects, an additional syntax rule is used to assign a newly created object to a variable. Let $x \in \mathsf{Names}$ be the name of a variable, $c \in \mathsf{Names}$ be the name of a class, and $\overline{e} \equiv e_1, \ldots, e_n \in \overline{\mathsf{Exp}}$ be a list of expressions. New instances of class C may be created as follows: $x :=$ `new` $C(\overline{e})$. The expressions $e_1, \ldots, e_n$ are assigned to the class parameters specified in the class declaration. If the class has a parameterless method called `init`, this method executes immediately. If class C has superclasses, the superclasses' `init` methods are called recursively, before C's version of `init` is run. In addition, if a `run` method is declared or inherited by the class, it is invoked immediately after `init`. Remark that `init` implicitly gets its parameters from the class and that `run` does not have parameters.

The statement **release** represents unconditional release. The **await** statement is used for conditional processor release. The statement **await** g releases the processor if the guard g evaluates to false and reacquires the processor at some later time when g is true. The syntactic category Guard of *conditional guards* is constructed by the production g in Figure 1, including (conjunctions and disjunctions of) Boolean expressions b over the local state of the object and queries $t?$ of whether replies to method calls made by the current process have arrived.

Any method may be called synchronously with the syntax $o.m(\overline{e};\overline{y})$ or asynchronously with the syntax $l!o.m(\overline{e});\ \ldots;\ l?(\overline{y})$. Here, $l \in \mathsf{Names}$ is a label, i.e. a special kind of variable used to identify an asynchronous call, $o \in \mathsf{Exp}$ is an object expression of type t, $m \in \mathsf{Names}$ is the name of a method supported by the t type, $\overline{e} \in \overline{\mathsf{Exp}}$ is a list of input arguments, and $\overline{y} \in \overline{\mathsf{Names}}$ is a list of output arguments. As usual, the input and output arguments must match the parameters declared in the method's signature. In addition, the calling object must implement the method's cointerface if one was specified.

Asynchronous method calls consist of an invocation and a reply. The invocation can be seen as a message from the caller to the called method, with arguments corresponding to the method's input parameters. The reply is a message from the called method, containing the return values for the call. Thus, in the asynchronous case, the call is captured by two separate statements; an invocation $l!o.m(\overline{e})$ and a reply $l?(\overline{y})$. Here, $l \in \mathsf{Names}$ is a label whose value identifies the asynchronous call. This label makes it possible to refer to a specific asynchronous call in cases where several calls are active at the same time.

The guard $l?$ evaluates to true if and only if a reply for the call identified by label l is available, and allows us to program non-blocking calls. In particular, the sequence $t!o.m(\overline{e});\mathbf{await}\ t?;t?(\overline{y})$ gives a standard non-blocking call, and is abbreviated **await** $o.m(\overline{e};\overline{y})$.

Example 4.1 The following code initiates an asynchronous method call, executes some statements, releases the processor if the reply has not arrived, and finally retrieves the return values stored in the reply:

```
var result: Int; var l: Label[Int];
  l!server.request(); ...; await l?; l?(result)
```

Without the **await** statement, the program would block on the reply statement `l?(result)` until the associated method invocation had terminated. This is the standard behavior of transparent futures (e.g., [11]), in which case the reply statement could be made implicit by identifying `l` with **reply**.

Method calls may be local or remote. Local synchronous calls correspond to the case where o is omitted or evaluates to **self** and are executed immediately in order to avoid deadlocking the object. In contrast, a remote synchronous call $o.m(\overline{e};\overline{y})$ is implemented as an asynchronous method invocation $t!o.m(\overline{e})$ followed by the reply statement $t?(\overline{y})$, for some fresh label t. Because there is no **await** statement between the method invocation and the reply statement, the calling object is blocked while the remote method executes. The values assigned to the output parameters become available in $\overline{y}$.

When overriding a method in a subclass, it is often necessary to call the original superclass implementation of the method as well. This is done using *qualified method calls*, which have the syntax $l!m@C(\overline{e})$ for the local asynchronous call, $m@C(\overline{e};\overline{y})$ for the local synchronous call, and **await** $m@C(\overline{e};\overline{y})$ for the local non-blocking call, where $C \in \mathsf{Names}$ is the name of a superclass. Qualified method calls give static access to methods defined in any superclass and are always local.

4.4 Compound Statements

Statements can be grouped (s) and composed using sequential composition $s_1; s_2$, conditionals **if** b **then** s_1 **else** s_2 **end**, and loops **while** b **do** s **end**, with $b \in \mathsf{BoolExp}$ and $s, s_i \in \texttt{Stmt}$. In addition, Creol offers an operator for nondeterministic choice, written $S_1 \,\square\, S_2$. The composition operator ; binds more strongly than the choice operator $\square$ and both operators are associative.

In order to understand the choice operator, the following definitions are used. A statement S is *enabled* if it can execute in the current state without releasing the processor immediately. For example, **await** b is enabled if b evaluates to true; otherwise it is disabled. If s is enabled and does not block immediately, then we say that s is *ready*. The statement $l?(\overline{y})$ is always enabled, but it is only ready if the reply associated with l has arrived.

The nondeterministic choice statement $s_1 \,\square\, s_2$ executes either s_1 or s_2. If both branches s_i are ready, then $s_1 \,\square\, s_2$ chooses either s_1 or s_2. If only one branch s_i is ready, that branch is executed. If neither branch is ready, $s_1 \,\square\, s_1$ blocks if either s_1 or s_2 is enabled and releases the processor otherwise.

Example 4.2 The nondeterministic statements are typically used in conjunction with asynchronous method calls. Consider the following method bodies:

```
var res: Int; var l1, l2: Label[Int];
  l1!server1.request(); l2!server2.request();
    l1?(res) □ l2?(res); processResult(res)

var res1,res2: Int; var l1, l2: Label[Int];
  l1!server1.request(); l2!server2.request();
    l1?(res1); processResult(res1); l2?(res); processResult(res2)
      □ l2?(res2); processResult(res2); l1?(res1); processResult(res1)
```

In both method bodies, two asynchronous method calls are made: one to `server1` and one to `server2`. In the first method body the nondeterministic choice selects the first reply which arrives for further processing, and the other reply is ignored. In the second method body, both replies are processed, but the order of processing depends on which reply arrives first. (The second method body suggests a nondeterministic merge operator, discussed in [30].)

4.5 The Full Language and Formal Semantics

In this paper, focus has been on the basic model of concurrency and communication in Creol. A more in-depth discussion of this model and its formal semantics may be found in [30]. The integration of first-class futures in Creol is discussed in [17]. An industrial case study of the ASK communication system in Creol is presented in [4].

Type systems for asynchronous method calls are developed in [36] and for the full language in [35]. An extension of Creol for service-oriented computing is presented in [13] and for wireless communication in [32]. Creol further supports runtime reprogramming of distributed systems by means of a dynamic class construct which allows objects of a class and its subclasses to gradually evolve in a type-safe way without interrupting the overall system execution [33,42].

5 Analysis of Creol Models

Creol is a formally defined language with an operational semantics defined in rewriting logic [38]. Rewriting logic specifications are executable on the Maude rewriting engine [14], which provides a range of analysis facilities like simulation, search, and model checking. Thus Maude works as an interpreter for Creol models with support for analysis. A compiler and type checker for Creol is available which translates Creol models to the input format of the interpreter. A plug-in to Eclipse helps in developing models and which interacts with the interpreter; e.g., it is possible to query a given state of an execution without being exposed to the underlying Maude representation of the runtime state. The Creol compiler and interpreter, as well as the Eclipse front-end and a range of example models, are available from the Creol web-site [15].

Creol models of distributed systems are highly nondeterministic, both with respect to intra-object scheduling and to inter-object communication. Testing methods for Creol models exploit the flexibility of the Maude rewriting engine to drive the interpreter in specific directions. In order to customize a concurrent object for a specific application, a scheduler may be added to the object which resolves aspects of its nondeterminism. Initial work on testing concurrent objects with application-specific schedulers is presented in [40]. Inter-object testing exploits the strong encapsulation of the internal object state in Creol and is therefore based on observable behavior, building on a theory of observability for distributed concurrent objects [1]. Using Maude's support for reflection, the monitoring of communication in the interpreter may be done in a non-obtrusive way, leaving the language semantics unchanged. However, rule selection may be guided by predicates expressing constraints on the observable behavior of objects, partly resolving inter-object nondeterminism [34,25].

The concurrency model and strong encapsulation of Creol is also exploited in the proof system for the language. Reasoning about Creol models is done in two steps. First the intra-object behavior is specified in terms of a class invariant, using an auxiliary variable for the local history of communication. This invariant is similar to a monitor invariant, but in contrast to monitor reasoning we need not be concerned with the amount of signaling between processes as this is taken care of by the semantics of processor release points. The proof system for internal reasoning about concurrent objects with asynchronous method calls and futures is based on purely local proof rules and is sound and complete [17]. The relationship between model interpretation in Maude and the proof system is studied in [8]. Since class inheritance and interface subtyping are distinct, code reuse can be more flexible than with behavioral subtyping [37]. Nevertheless, code reuse supports incremental reasoning based on lazy behavioral subtyping [22,23]. Based on the class invariant,

the behavioral specification of the interfaces implemented by the class need to be derived [21]. Consequently, reasoning about object composition is done in terms of observable behavior, relying on the behavioral properties of the interfaces, specifying the observable object behavior.

6 Conclusion

Creol is an object-oriented language that integrates high-level programming constructs for handling inter-object and intra-object concurrency. The language guarantees that an active method has exclusive access to the object's fields, as in a monitor. Each object has its own virtual processor, and pending method invocations on the same object compete for the processor. By enforcing mutual exclusion within an object and by relying on explicit processor release points, the interference issue related to shared variable concurrency in multithreaded object-oriented languages is avoided. Objects communicate with each other through asynchronous method calls, which consist of an invocation and a reply message. This allows the calling object to perform other activities while the call is being serviced. Using nondeterministic statements, the caller can process replies in the order in which they arrive, allowing execution to adapt to the environment in a flexible way. Interfaces with cointerface requirements statically control communication, enabling call-back between peer objects in the distributed setting.

The Creol modeling language is executable on an interpreter which supports query-driven resolution of nondeterminism based on the runtime state of execution. This enables the application of a range of testing methods to the language and experimentation with application-specific scheduling at the modeling stage. Furthermore, the concurrency model and object encapsulation adopted in the modeling language enables compositional reasoning about concurrent distributed systems.

Acknowledgments. Many people have contributed to the development of the Creol language and tools. Among the work reported in this paper, we are specifically grateful for the contributions of Bernhard Aichernig, Joakim Bjørk, Frank de Boer, Dave Clarke, Johan Dovland, Andreas Griesmayer, Rudi Schlatte, Andries Stam, Martin Steffen, Arild Torjusen, and Ingrid Chieh Yu.

References

[1] Ábrahám, E., I. Grabe, A. Grüner and M. Steffen, *Abstract interface behavior of an object-oriented language with futures and promises* (2008), submitted as invited journal contribution to a special edition of the Journal of Logic and Algebraic Programming (NWPT'07).

[2] Ábrahám-Mumm, E., F. S. de Boer, W.-P. de Roever and M. Steffen, *Verification for Java's reentrant multithreading concept*, in: M. Nielsen and U. H. Engberg, editors, *FoSSaCS*, Lecture Notes in Computer Science **2303** (2002), pp. 4–20.

[3] Agha, G. A., I. A. Mason, S. F. Smith and C. L. Talcott, *A foundation for actor computation*, Journal on Functional Programming **7** (1997), pp. 1–72.

[4] Aichernig, B., A. Griesmayer, R. Schlatte and A. Stam, *Modeling and testing multi-threaded asynchronous systems with Creol*, in: *Proceedings of the 2nd International Workshop on Harnessing Theories for Tool Support in Software (TTSS'08)*, Electronic Notes in Theoretical Computer Science (2008), this volume.

[5] Andrews, G. A., "Foundations of Multithreaded, Parallel, and Distributed Programming," Addison-Wesley, 1999.

[6] Baker, H. G. and C. E. Hewitt, *The incremental garbage collection of processes*, ACM SIGPLAN Notices **12** (1977), pp. 55–59.

[7] Benton, N., L. Cardelli and C. Fournet, *Modern concurrency abstractions for C♯*, ACM Transactions on Programming Languages and Systems **26** (2004), pp. 769–804.

[8] Blanchette, J. C. and O. Owe, *An open system operational semantics of an object-oriented and component based language*, in: *Proc. 4th International Workshop on Formal Aspects of Component Software*, Electronic Notes in Theoretical Computer Science **215** (2008), pp. 151–169.

[9] Brinch Hansen, P., *The nucleus of a multiprogramming system*, CACM **13** (1970), pp. 238–241.

[10] Brinch Hansen, P., *Java's insecure parallelism*, ACM SIG-PLAN Notices **34** (1999), pp. 38–45.

[11] Caromel, D. and L. Henrio, "A Theory of Distributed Object," Springer-Verlag, 2005.

[12] Cenciarelli, P., A. Knapp, B. Reus and M. Wirsing, *An event-based structural operational semantics of multi-threaded Java*, in: J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, Lecture Notes in Computer Science **1523** (1999), pp. 157–200.

[13] Clarke, D., E. B. Johnsen and O. Owe, *Concurrent objects à la carte*, in: D. Dams, U. Hannemann and M. Steffen, editors, *Correctness, Concurrency, Compositionality: Essays in honor of Willem-Paul de Roever*, Lecture Notes in Computer Science (2008), in preparation.

[14] Clavel, M., 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** (2002), pp. 187–243.

[15] Creol homepage, http://www.ifi.uio.no/~creol.

[16] Cugola, G. and C. Ghezzi, *Cjava: Introducing concurrent objects in Java*, in: M. E. Orlowska and R. Zicari, editors, *OOIS 1997* (1997), pp. 504–514.

[17] de Boer, F. S., D. Clarke and E. B. Johnsen, *A complete guide to the future*, in: R. de Nicola, editor, *Proc. 16th European Symposium on Programming (ESOP'07)*, Lecture Notes in Computer Science **4421** (2007), pp. 316–330.

[18] de Roever, W.-P., F. S. de Boer, U. Hannemann, J. Hooman, Y. Lakhnech, M. Poel and J. Zwiers, "Concurrency Verification: Introduction to Compositional and Noncompositional Methods," Number 54 in Cambridge Tracts in Theoretical Computer Science, Cambridge University Press, 2001.

[19] Di Blasio, P. and K. Fisher, *A calculus for concurrent objects*, in: U. Montanari and V. Sassone, editors, *7th International Conference on Concurrency Theory (CONCUR'96)*, Lecture Notes in Computer Science **1119** (1996), pp. 655–670.

[20] Dovland, J., E. B. Johnsen and O. Owe, *Verification of concurrent objects with asynchronous method calls*, in: *IEEE International Conference on Software - Science, Technology and Engineering (SwSTE'05)* (2005), pp. 141–150.

[21] Dovland, J., E. B. Johnsen and O. Owe, *Observable behavior of dynamic systems: Component reasoning for concurrent objects*, in: D. Goldin and F. Arbab, editors, *Proc. Workshop on the Foundations of Interactive Computation (FInCo'07)*, Electronic Notes in Theoretical Computer Science **203** (2008), pp. 19–34.

[22] Dovland, J., E. B. Johnsen, O. Owe and M. Steffen, *Lazy behavioral subtyping*, in: J. Cuellar and T. Maibaum, editors, *Proc. 15th International Symposium on Formal Methods (FM'08)*, Lecture Notes in Computer Science **5014** (2008), pp. 52–67.

[23] Dovland, J., E. B. Johnsen, O. Owe and M. Steffen, *Incremental reasoning for multiple inheritance*, in: *Proc. 10th International Conference on Integrated Formal Methods (IFM'09)*, Lecture Notes in Computer Science (2009), to appear.

[24] Gosling, J., B. Joy and G. L. Steele, "The Java Language Specification," Addison-Wesley, 2005, 3rd edition.

[25] Grabe, I., M. Kyas, M. Steffen and A. B. Torjusen, *Executable interface specifications for testing asynchronous Creol components*, Technical Report 375, Dept. of Informatics, University of Oslo (2008), submitted.

[26] Hewitt, C., *Viewing control structures as patterns of passing messages*, Technical Report 410, Massachusetts Institute of Technology, Artificial Intelligence Laboratory (1976).

[27] Hoare, C. A. R., *Monitors: An operating system structuring concept*, CACM **17** (1974), pp. 549–557.

[28] International Telecommunication Union, Geneva, "Open Distributed Processing - Reference Model parts 1–4," (1995).

[29] Johnsen, E. B. and O. Owe, *A compositional formalism for object viewpoints*, in: B. Jacobs and A. Rensink, editors, *Formal Methods for Open Object-Based Distributed Systems V* (2002), pp. 45–60.

[30] Johnsen, E. B. and O. Owe, *An asynchronous communication model for distributed concurrent objects*, Software and Systems Modeling **6** (2007), pp. 35–58.

[31] Johnsen, E. B., O. Owe and M. Arnestad, *Combining active and reactive behavior in concurrent objects*, in: D. Langmyhr, editor, *Proc. Norwegian Informatics Conference* (2003), pp. 193–204.

[32] Johnsen, E. B., O. Owe, J. Bjørk and M. Kyas, *An object-oriented component model for heterogeneous nets*, in: F. S. de Boer, M. M. Bonsangue, S. Graf and W.-P. de Roever, editors, *Proc. 6th International Symposium on Formal Methods for Components and Objects (FMCO 2007)*, Lecture Notes in Computer Science **5382** (2008), pp. 257–279.

[33] Johnsen, E. B., O. Owe and I. Simplot-Ryl, *A dynamic class construct for asynchronous concurrent objects*, in: M. Steffen and G. Zavattaro, editors, *Proc. 7th International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS'05)*, Lecture Notes in Computer Science **3535** (2005), pp. 15–30.

[34] Johnsen, E. B., O. Owe and A. B. Torjusen, *Validating behavioral component interfaces in rewriting logic*, Fundamenta Informaticae **82** (2008), pp. 341–359.

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

[36] Johnsen, E. B. and I. C. Yu, *Backwards type analysis of asynchronous method calls*, Journal of Logic and Algebraic Programming **77** (2008), pp. 40–59.

[37] Liskov, B. H. and J. M. Wing, *A behavioral notion of subtyping*, ACM Transactions on Programming Languages and Systems **16** (1994), pp. 1811–1841.

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

[39] Nierstrasz, O., *Active objects in Hybrid*, in: N. Meyrowitz, editor, *OOPSLA 1987* (1987), pp. 243–253.

[40] Schlatte, R., B. Aichernig, F. de Boer, A. Griesmayer and E. B. Johnsen, *Testing concurrent objects with application-specific schedulers*, in: J. S. Fitzgerald, A. E. Haxthausen and H. Yenigun, editors, *Proc. 5th International Colloquium on Theoretical Aspects of Computing (ICTAC'08)*, Lecture Notes in Computer Science **5060** (2008), pp. 319–333.

[41] Yonezawa, A., "ABCL: An Object-Oriented Concurrent System," MIT, 1990.

[42] Yu, I. C., E. B. Johnsen and O. Owe, *Type-safe runtime class upgrades in Creol*, in: R. Gorrieri and H. Wehrheim, editors, *Proc. 8th International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS'06)*, Lecture Notes in Computer Science **4037** (2006), pp. 202–217.