A Dynamic Class Construct for Asynchronous Concurrent Objects

Einar Broch Johnsen[1], Olaf Owe[1], and Isabelle Simplot-Ryl[2]

[1] Department of Informatics, University of Oslo, Norway
{einarj,olaf}@ifi.uio.no
[2] LIFL, CNRS-UMR 8022, University of Lille I, France
ryl@lifl.fr

Abstract Modern applications distributed across networks such as the Internet may need to evolve without compromising application availability. Object systems are well suited for runtime upgrade, as encapsulation clearly separates internal structure and external services. This paper considers a mechanism for dynamic class upgrade, allowing class hierarchies to be upgraded in such a way that the existing objects of the upgraded class and of its subclasses gradually evolve at runtime. The mechanism is integrated in Creol, a high-level language which targets distributed applications by means of concurrent objects communicating by asynchronous method calls. The dynamic class construct is given a formal semantics in rewriting logic, extending the semantics of the Creol language.

1 Introduction

For critical distributed applications, which are long lived and have high availability requirements, it is important that system components can be upgraded in response to new requirements that arise over time without compromising application availability. Requirements that necessitate component upgrade may be additional features and improved performance, as well as bugfixes. Examples of such applications are found in banks, air traffic control systems, aeronautics, financial transaction processes, and mobile and Internet applications. For these systems, manual reconfiguration and recompilation of components are both impractical, due to component distribution, and unsatisfactory, due to the high availability requirements. Instead, upgrades and patches should be applicable at runtime. Early approaches to software distribution and upgrading [2, 4, 6, 14, 16, 21, 26] do not address the need for continuous availability during upgrades. More recently, the issue of runtime reconfiguration and upgrade has attracted attention [1, 3, 5, 7, 11, 13, 15, 22, 25, 27]. For large distributed systems, it seems desirable to perform upgrades in an asynchronous and modular way, such that upgrades propagate automatically through the distributed system. An automatic upgrade system should [1,27]: propagate upgrades automatically, provide a means to control *when* components may be upgraded, and ensure the availability of system services even in the course of an upgrade process, when components of different versions coexist.

In this paper, we propose and formalize a solution to these issues, taking an object-oriented approach. Our solution is based on a dynamic class construct, allowing class definitions to be upgraded at runtime. Upgrading a class affects all future instances of

In: M. Steffen and G. Zavattaro (Eds.):
Proc. 7th Intl. Conf. on Formal Methods for Open Object-Based Distributed Systems
(FMOODS'05), p. 15–30, LNCS 3535, Springer-Verlag, June 2005.

the redefined class and of its subclasses. Further, all *existing* object instances of the class and its subclasses are upgraded. In contrast to e.g. [1], our approach is completely distributed, and no centralized versioning repository is required. No specific measures are needed by the programmer to anticipate and prepare for future upgrades, as the upgrade process itself is handled transparently by the runtime system. Whereas [5, 9, 10, 29] present formal systems for upgrades of sequential languages or modifications to single objects, we are not aware of any formalization of modular upgrades for concurrent object systems. In contrast to all the cited works, our approach addresses and exploits *inheritance*, supports synchronous as well as *asynchronous* communication, and allows both *nonterminating*, *active*, and reactive processes in objects to be upgraded.

This paper considers dynamic class upgrades in the Creol language [17–19], which specifically targets open distributed systems with concurrent objects, has multiple inheritance, and supports both asynchronous and synchronous invocation of object methods. Creol has an operational semantics defined in rewriting logic [23] and an interpreter running on the Maude platform [8, 23]. In this paper, a dynamic class construct is proposed and formalized in rewriting logic through integration in Creol's operational semantics.

Paper overview. Sect. 2 summarizes the Creol language and presents the dynamic class construct, Sect. 3 provides two examples, Sect. 4 presents the operational semantics, Sect. 5 discusses related work, and Sect. 6 concludes the paper.

2 A Language for Asynchronously Communicating Objects

This section briefly reviews the basic features of Creol [17–19], a high-level language for distributed concurrent objects. We distinguish data, typed by data types, and objects, typed by interfaces. The language allows both synchronous and asynchronous invocation of methods, based on a uniform semantics. Attributes (object variables) and method declarations are organized in classes, which may have data and object parameters. Concurrent objects have their own processor which evaluates local processes, i.e. program code with *processor release points*. Processes may be *active*, reflecting autonomous behavior initiated at creation time by the *run* method, or *reactive*, i.e. in response to method invocations. Due to processor release points, the evaluation of processes may be interleaved. The values of an object's program variables may depend on the non-deterministic interleaving of processes. Therefore a method instance may have local variables supplementing the object variables, in particular the values of formal parameters are stored locally. An object may contain several (pending) instances of the same method, possibly with different values for local variables.

2.1 Asynchronous and Synchronous Method Invocations

All object interaction happens through method calls. A method may be invoked either synchronously or asynchronously [17]. When a process invokes a method asynchronously, the process may continue its activity until it requests a reply to the call or it is suspended by arriving at a processor release point in its code. In the asynchronous setting method calls can always be emitted, as a receiving object cannot block communication. *Method overtaking* is allowed: if methods offered by an object are invoked in one order, the object may start evaluation of the method instances in another order.

An asynchronous method call is made with the statement $t!x.m(\mathrm{E})$, where $t \in \mathsf{Label}$ provides a locally unique reference to the call, x is an object expression, m a method name, and E an expression list with the actual parameters supplied to the method. Labels identify invocations and may be omitted if a reply is not explicitly requested. Return values from the call are explicitly fetched, say in a variable list V, by the statement $t?(\mathrm{V})$. This statement treats V as a list of future variables [30]: If a reply has arrived, return values are assigned to V and evaluation continues. In the case of a local call, i.e. when the value of x is the same as *this* object, the processor is released to start evaluation of the call. Otherwise, process evaluation is blocked. In order to avoid blocking in the asynchronous case, *processor release points* are introduced for reply requests (Sect. 2.2): If no reply has arrived, evaluation is *suspended* rather than blocked.

Synchronous (RPC) method calls, immediately blocking the processor while waiting for a reply, are written $p(\mathrm{E};\mathrm{V})$; this is shorthand for $t!p(\mathrm{E});t?(\mathrm{V})$, where t is a fresh label variable. The language does not support monitor reentrance (except for local calls), mutual synchronous calls may therefore lead to deadlock. In order to evaluate local calls, the invoking process must eventually suspend its own evaluation. In particular, the evaluation of synchronous local calls will precede the active process. Local calls need not be prefixed by an object identifier, in which case they may be identified syntactically as internal. The keyword *this* is used for self reference in the language.

2.2 Processor Release Points

Guards g in statements **await** g explicitly declare potential processor release points. When a guard which evaluates to false is encountered during process evaluation, the process is suspended and the processor released. After processor release, any pending process may be selected for evaluation. The type Guard is defined inductively:

- $wait \in \mathsf{Guard}$ (explicit release),
- $t? \in \mathsf{Guard}$, where $t \in \mathsf{Label}$,
- $b \in \mathsf{Guard}$, where b is a boolean expression over local and object state,
- $g_1 \wedge g_2$ and $g_1 \vee g_2$, where $g_1, g_2 \in \mathsf{Guard}$.

Use of *wait* will always release the processor. The reply guard $t?$ is enabled if a reply to the call with label t has arrived. Evaluation of guard statements is atomic. We let **await** $g \wedge t?(\mathrm{V})$ abbreviate **await** $g \wedge t?$; $t?(\mathrm{V})$ and we let **await** $p(\mathrm{E};\mathrm{V})$, where p is a method call (external or internal), abbreviate $t!p(\mathrm{E})$; **await** $t?(\mathrm{V})$ for some fresh label t.

Statements can be composed to reflect requirements to the internal object control flow. Let S_1 and S_2 denote statement lists. An unguarded statement list is always enabled. Sequential composition may introduce guards: **await** g is a potential release point in S_1; **await** g; S_2. Nondeterministic choice $\mathrm{S}_1 \Box \mathrm{S}_2$ may select S_1 once S_1 is enabled or S_2 once S_2 is enabled, and suspends if neither branch is enabled. Nondeterministic merge $\mathrm{S}_1 ||| \mathrm{S}_2$ evaluates the statements S_1 and S_2 in some interleaved and enabled order. In addition there are standard constructs for if-statements and internal method calls, including recursive calls. Note that for the purposes of dynamic upgrades, recursive calls replace while-loops in the language. Assignment to local and object variables is expressed as $\mathrm{V} := \mathrm{E}$ for a disjoint list of program variables V and an expression list E, of matching types. In-parameters as well as *this*, *label*, and *caller* are read-only variables.

Syntactic categories.	*Definitions.*
g in Guard	$g ::= \mathit{wait} \mid b \mid t? \mid g_1 \wedge g_2 \mid g_1 \vee g_2$
p in MtdCall	$p ::= x.m \mid m@\mathit{classname} \mid m$
s in Stm	$\mathrm{S} ::= s \mid s; \mathrm{S}$
t in Label	$s ::= \mathbf{skip} \mid (\mathrm{S}) \mid \mathrm{S}_1 \Box \mathrm{S}_2 \mid \mathrm{S}_1 \mathbin{\vert\vert\vert} \mathrm{S}_2$
v in Var	$\mid \mathrm{V} := \mathrm{E} \mid v := \mathbf{new}\ \mathit{classname}(\mathrm{E})$
e in Expr	$\mid \mathbf{if}\ b\ \mathbf{then}\ \mathrm{S}_1\ \mathbf{else}\ \mathrm{S}_2\ \mathbf{fi}$
x in ObjExpr	$\mid t!p(\mathrm{E}) \mid !p(\mathrm{E}) \mid p(\mathrm{E};\mathrm{V}) \mid t?(\mathrm{V})$
b in Bool	$\mid \mathbf{await}\ g \mid \mathbf{await}\ g \wedge t?(\mathrm{V}) \mid \mathbf{await}\ p(\mathrm{E};\mathrm{V})$
m in Mtd	

Figure 1. An outline of the language syntax for method definitions, with typical terms for each category. Capitalized terms such as S, V, and E denote lists, sets, or multisets of the given syntactic categories, depending on the context.

With release points, the object need not block while waiting for replies. This approach is more flexible than future variables: suspended processes or new method calls may be evaluated while waiting. If the called object never replies, deadlock is avoided as other activity in the object is possible. However, when the reply arrives, the *continuation* of the process must compete with other pending and enabled processes.

2.3 Multiple Inheritance and Virtual Binding

The Creol language provides a mechanism for multiple inheritance [18] where all attributes and methods of a superclass are inherited by the subclass, and where superclass methods may be redefined. Class inheritance is declared by a keyword **inherits** which takes as argument an *inheritance list*; i.e., a list of class names $C(\mathrm{E})$ where E provides the actual class parameters. We say that a method or attribute is defined *above* a class C if it is declared in C or in at least one of the classes inherited by C. Internal calls are executed on the caller and may therefore take advantage of the statically known class structure to invoke specific method declarations. We introduce the syntax $t!m@C(\mathrm{E})$ for asynchronous and $m@C(\mathrm{E};\mathrm{V})$ for synchronous internal invocation of a method above C in the inheritance graph from C or a subclass of C. These calls may be bound without knowing the exact class of *this* object , so they are called *static*. In contrast calls without @, called *virtual*, need to identify the actual class of the callee at runtime in order to bind the call. We assume that attributes have unique names in the inheritance graph; this may easily be enforced at compile time and implies that attributes are bound statically. Consequently, a method declared in a class C may only access attributes declared above C. In a subclass, an attribute x of a superclass C is accessed by the qualified reference $x@C$. The language syntax is given in Fig. 1.

Virtual binding. When a method is virtually invoked in an object o of class C, a method declaration is identified in the inheritance graph of C and bound to the call. For simplicity, the call is bound to the first matching method definition above C in the inheritance graph, in a left-first depth-first order. Assume given a nominal subtype relation as a reflexive partial ordering $\prec$ on types, including interfaces. A data type may only be a subtype of a data type and an interface may only be a subtype of an interface. If $T \prec T'$ then any value of T may masquerade as a value of T'. Subtyping for type tuples is the pointwise extension of the subtype relation: $T \prec T'$ if the tuples T and T' have the same length l and $T_i \prec T_i'$ for every i ($0 \leq i \leq l$) and types T_i and T_i' in position i in T and T'. To explain the typing and binding of methods, subtyping is extended to function spaces $A \rightarrow B$, where A and B are (possibly zero-length) type tuples:

$$A \rightarrow B \prec A' \rightarrow B' = A \prec A' \wedge B' \prec B.$$

The static analysis of an internal call $m(\mathrm{E};\mathrm{V})$ will assign unique types to the in- and out-parameter depending on the textual context. Say that the actual parameters are textually declared as $\mathrm{E} : T_{\mathrm{E}}$ and $\mathrm{V} : T_{\mathrm{V}}$. The call is *type correct* if there is a method declaration $m : A \rightarrow B$ above the class C such that $T_{\mathrm{E}} \rightarrow T_{\mathrm{V}} \prec A \rightarrow B$. The binding of an asynchronous call $t!m(\mathrm{E})$ with a reply $t?(\mathrm{V})$ or **await** $t?(\mathrm{V})$, is handled as the corresponding synchronous call $m(\mathrm{E};\mathrm{V})$.

At runtime the object making the internal call $m : T_{\mathrm{E}} \rightarrow T_{\mathrm{V}}$ will be of a subclass C' of C and the virtual binding mechanism will bind to a declaration of $m : A' \rightarrow B'$ such that $T_{\mathrm{E}} \rightarrow T_{\mathrm{V}} \prec A' \rightarrow B'$, taking the first such m above C'. Because C is inherited by C', the virtual binding is guaranteed to succeed. External calls $t!o.m(\mathrm{E})$ are virtually bound in the graph above the dynamically identified class of o. Provided that the declared interface of o supports the method signature, successful binding is guaranteed for any instance of a type-correct class implementing the interface.

2.4 System Evolution Through Class Upgrade

System change is addressed through a mechanism for class upgrade, which allows existing and future objects of the upgraded class and of its subclasses to evolve. A class may be subjected to a number of upgrades. In an upgrade, new attributes, methods, and superclasses may be added to a class definition, and old methods may be modified. In order to allow old method instances to evaluate safely and avoid runtime type errors, no attributes, methods, or inherited classes may be removed as part of a class upgrade. Although more restrictive, empirical studies suggest that addition and redefinition of services are far more common forms of software evolution than removal [29].

Attributes may be added. New attributes may be added to a class. The addition of a new attribute with the same name as another attribute already defined in the class is not allowed. The addition of an attribute having the same name as an inherited attribute is allowed. The instance of the class will then have both attributes, which are accessed by qualified names (see Sect. 2.3). As attribute names are statically expanded into qualified names, old code will continue to use the same attributes as before the upgrade.

Methods may be added or redefined. We consider the effect of adding or redefining a method in a class C with respect to the sub- and superclasses of C. If a method is *redefined* in C, the method's code is replaced in all instances of C and the old method definition is no longer available. This leads to a *subtyping discipline* for method redefinitions in order to ensure that virtual binding succeeds. Consequently, we allow a method's internal data structures to be replaced, but for redefinition covariance and contravariance is required for the method's in- and out-parameters, respectively.

If a method is *added* to a class, virtual binding guarantees that old calls are type correct without placing any restrictions on the new method. All kinds of overloading of inherited methods are allowed, including overloading with respect to the number of in- or out-parameters. For method declarations with the same number of in- and out-parameters overloading may be with respect to parameter types, possibly only for out-parameters. If a method m is added to C and m is previously defined in a superclass C' of C, the new definition in C will override (and hide) the inherited method m of C' in the sense that a call which matches both definitions will be bound differently after the upgrade. The superclass method is still available by the static call $m@C'$. Virtual binding ensures that calls that were type correct before the class upgrade remain type correct. If a method m is added to C and m is previously defined in a subclass C'' of C, a new override relationship will be introduced. However, virtual binding preserves the type correctness of old calls as well as the virtually bound calls of the upgraded class. The addition of a method to a class C does not need to be restricted by definitions in the sub- or superclasses of C.

Superclasses may be added. If a class C is added as a superclass during a class upgrade, the attributes and methods defined in C and its superclasses become available. The binding mechanism works in a left-first depth-first order, so the order of the list of inherited classes is crucial: to minimize the effect of new superclasses on the virtual binding mechanism, the new superclasses are added at the end of the inheritance list.

In order to avoid runtime errors in the case when old code contains calls to the *new* method with the old parameter list, we do not allow the formal parameter list of a class to be extended. (It is straightforward to avoid this restriction using default values.) In addition we do not allow the types of formal parameters to change; wider types could create errors for old code operating on new objects whereas narrower types could create errors for new code operating on old objects. Consequently, the actual parameters to the new superclasses must be expressed by means of the old class parameters and attributes.

3 Examples

Two examples of dynamic upgrade are considered. Upgrade is used to add a new service to an existing class, with visible effects to users of this class, and to change a communication protocol at runtime, to increase the system performance in a transparent manner.

3.1 Example: A Bank Account

Consider a bank account of interface *Account*, with methods for deposit and transfer of funds, such that a transfer must wait until the account has sufficient funds.

```
class BankAccount implements Account                                --- Version 1
begin var bal : Int = 0
with Any
   op deposit (in sum : Nat) == bal := bal+sum
   op transfer (in sum : Nat, acc : Account) ==
         await bal ≥ sum ; bal := bal−sum; acc.deposit(sum)
end
```

Dynamic class upgrade allows the addition of new services to such an application without stopping the system. Let us consider the addition of overdraft facilities. The upgrade of the *BankAccount* class will add a method *overdraft_open* such that an object supporting the *Banker* interface may set a maximal overdraft amount for the account. The *transfer* method will be upgraded to take this new facility into account.

```
upgrade class BankAccount                                  --- Version 2
begin var overdraft : Nat = 0
with Any
  op transfer (in sum : Nat, acc : Account) ==
      await bal ≥ sum−overdraft; bal := bal−sum; acc.deposit(sum)
with Banker
  op overdraft_open (in max : Nat) == overdraft := max
end
```

An *overdraft* variable is added during upgrade. Pending transfer processes in an object await the old guard, whereas new transfer calls to the object get the new guard.

3.2 Example: Broadcast in Ad Hoc Networks

Consider a wireless broadcast mechanism for ad hoc networks, using the *blind-flooding* protocol: When a node receives a message with a previously unseen sequence number, the message is sent to all neighbors and the sequence number is recorded.

```
class Node (neighbors : List[Oid])                         --- Version 1
begin var set : NatSet = emptySet
with Any
  op broadcast (in msg : Data, seqNbr : Nat) ==
     if seqNbr ∉ set then !neighbors.broadcast(msg, seqNbr); set := set ∪ {seqNbr} fi
end
```

The statement $!neighbors.m(\ldots)$ expands to a list of asynchronous calls to all elements in the *neighbors* list.

This protocol is “localized”: a node communicates with its direct neighbors and may ignore the overall network topology. However, there is a significant number of message collisions. Recently the *Neighbor Elimination Scheme* protocol has been introduced, which improves performance by reducing the total number of transmissions [28]. In the new protocol, a node knows both its neighbors and their neighbors. Intuitively, when a node receives a message it observes the communications for a certain time (using a timeout) and then decides not to resend the message if all of its neighbors have already received it. The system will now be upgraded to the new protocol at runtime.

We introduce a data type *Assoc* for sets of pairs to record the sequence numbers and sets of neighbors for the active broadcasts, with constructors $empty :\rightarrow Assoc$ and $add : Assoc \times Nat \times NatSet \rightarrow Assoc$. Define functions $isEmpty : Assoc \times Nat \rightarrow Bool$, $rem : Assoc \times Nat \times Oid \rightarrow Assoc$, and $remAll : Assoc \times Nat \rightarrow Assoc$ by the equations

$$\begin{aligned}
&\mathit{isEmpty}(\mathit{empty}, n) &&= \mathit{true}\\
&\mathit{isEmpty}(\mathit{add}(a,n,s),n') &&= \textbf{if } n = n' \textbf{ then } s = \emptyset \textbf{ else } \mathit{isEmpty}(a,n') \textbf{ fi}\\
&\mathit{rem}(\mathit{empty}, n, o) &&= \mathit{empty}\\
&\mathit{rem}(\mathit{add}(a,n,s),n',o) &&= \textbf{if } n = n' \textbf{ then } \mathit{add}(a,n,s \setminus \{o\}) \textbf{ else } \mathit{add}(\mathit{rem}(a,n',o),n,s) \textbf{ fi}\\
&\mathit{remAll}(\mathit{empty}, n) &&= \mathit{empty}\\
&\mathit{remAll}(\mathit{add}(a,n,s),n') &&= \textbf{if } n = n' \textbf{ then } \mathit{remAll}(a,n') \textbf{ else } \mathit{add}(\mathit{remAll}(a,n'),n,s) \textbf{ fi}
\end{aligned}$$

The *Node* class may now be upgraded:

```
upgrade Node                                                    --- Version 2
begin var a : Assoc = empty;
with Any
  op broadcast (in m : Data, n : Nat) == if n ∈ set then rem(a,n,caller)
    else set := set ∪ {n}; add(a,n,neighbors); rem(a,n,caller);
      await wait ∨ isEmpty(a,n);
        if ¬ isEmpty(a,n) then !neighbors.broadcast(m,n) else remAll(a,n) fi fi
end
```

Here the *set* and *neighbors* attributes are reused from the previous class version. The *wait* guard is used for delay, suspending the process for some amount of time. After the class upgrade, active *Node* objects are upgraded independently and at different times. There is a transitory period during which system performance gradually improves.

4 An Operational Semantics for Dynamic Class Upgrade

The operational semantics of Creol is defined in rewriting logic (RL) [23]. A rewrite theory is a 4-tuple $\mathcal{R} = (\Sigma, E, L, R)$ where the signature Σ defines the function symbols, E defines equations between terms, L is a set of labels, and R is a set of labeled rewrite rules. Rewrite rules apply to terms of given sorts. Sorts are specified in (membership) equational logic (Σ, E), the functional sublanguage of RL which supports algebraic specification in the OBJ [12] style. When modeling computational systems, different system components are typically modeled by terms of the different sorts defined in the equational logic. The global state configuration is defined as a multiset of these terms.

RL extends algebraic specification techniques with transition rules: The dynamic behavior of a system is captured by rewrite rules supplementing the equations which define the term language. From a computational viewpoint, a rewrite rule $t \longrightarrow t'$ may be interpreted as a *local transition rule* allowing an instance of the pattern t to evolve into the corresponding instance of the pattern t'. When auxiliary functions are needed in the semantics, these are defined in equational logic, and are evaluated in between the state transitions [23]. Rewrite rules apply to local fragments of a state configuration. If rewrite rules may be applied to nonoverlapping subconfigurations, the transitions may be performed in parallel. Consequently, concurrency is implicit in RL. Conditional rewrite rules are allowed, where the condition can be formulated as a conjunction of rewrites and equations which must hold for the main rule to apply:

$$\mathit{subconfiguration} \longrightarrow \mathit{subconfiguration} \textbf{ if } \mathit{condition}$$

A number of concurrency models have been successfully represented in RL [8,23], including Petri nets, CCS, Actors, and Unity, as well as the ODP computational model [24]. RL also offers its own model of object orientation [8], but inheritance in this model does not easily allow method overloading and redefinition. Rules in RL may be formulated at a high level of abstraction, closely resembling a compositional operational semantics.

System configurations. A Creol method call will be reflected by a pair of messages, and object activity will be organized around a *message queue* which contains incoming messages and a *process queue* which contains pending processes, i.e. remaining parts of method instances. In order to increase parallelism in the RL model, message queues will be external to object bodies. A state configuration is a multiset combining Creol objects, classes, messages, and queues. The associative constructor for lists is represented by ';', and the associative and commutative constructor for multisets by whitespace.

Objects in RL are commonly written as terms $\langle O : C \,|a_1 : v_1, \ldots, a_n : v_n\rangle$ where O is the object's identifier, C is its class, the a_i's are the names of the object's attributes, and the v_i's the corresponding values [8]. Adopting this form of presentation, we define Creol objects, classes, and external message queues as RL objects [17]. Omitting RL sorts, a Creol object is represented by an RL object $\langle Ob \,|\, Cl, Pr, PrQ, Lvar, Att, Lab\rangle$, where *Ob* is the object identifier, *Cl* the class name and *version number*, *Pr* the active process code, *PrQ* a multiset of pending processes with unspecified queue ordering, and *Lvar* and *Att* the local and object state, respectively. Let a sort τ be partially ordered by $<$, with least element 1, and let $Next : \tau \to \tau$ be such that $\forall x \,.\, x < Next(x)$. *Lab* is used to generate label values, which are terms of sort τ. Thus, the object identifier *Ob* and the generated local label value provide a globally unique identifier for each method call. A Creol object's message queue is represented as an RL object $\langle Qu \,|\, Ev\rangle$, where *Qu* is the queue identifier and *Ev* a multiset of unprocessed messages. Each message queue is associated with one specific Creol object. The statement $new\, C(\text{E})$ creates a new object (and associated queue) with a unique object identifier, object variables as listed in the class parameter list E and in *Att*, and places an instance of the *run* method in *Pr*.

Creol classes are represented by RL objects $\langle Cl \,|\, Inh, Att, Mtds, Tok\rangle$, where *Cl* is the class name and version number, *Inh* the inheritance list, *Att* a list of attributes, *Mtds* a multiset of methods, and *Tok* an arbitrary term of sort τ. Version n of a class named C will conventionally be denoted C # n. The rules for the static language constructs may be found in [17,18]. We shall here focus on the rules for dynamic class constructs.

4.1 Implicit Inheritance Graphs and Virtual Binding

In order to define dynamic reconfiguration mechanisms, the inheritance graph will not be statically given. Rather, the binding mechanism dynamically inspects the class hierarchy as present in the global state configuration. A *bind* message is sent from a class to its superclasses, resulting in a *bound* message returned to the object generating the *bind* message. This way, the inheritance graph is unfolded dynamically and as far as necessary when needed. This approach is used for virtual binding, and for collecting and instantiating the class variables of an object instance. We here present the virtual binding mechanism (see [18] for attribute collection). When the invocation $invoc(o, m, Sig, In)$ of a method m is found in the message queue of an object o of class

C, a message $bind(o,m,Sig,In,C)$ is generated where Sig is the method signature as provided by the caller and In is the list of actual in-parameters. Virtual calls are handled by the following rule:

$$\begin{array}{l}\langle o\!:\!Ob \,|\, Cl : C \;\#\; n\rangle \;\langle o\!:\!Qu \,|\, Ev : \textsc{q}\; invoc(o,m,Sig,In)\rangle \\ \quad \longrightarrow \langle o\!:\!Ob \,|\, Cl : C \;\#\; n\rangle \;\langle o\!:\!Qu \,|\, Ev : \textsc{q}\rangle \; bind(o,m,Sig,In,C \;\#\; n)\end{array}$$

Static method calls are generated by means of the same mechanism but without inspecting the *actual* class of the callee, thus surpassing local definitions:

$$\langle o\!:\!Qu \,|\, Ev : \textsc{q}\; invoc(o,m@C,Sig,In)\rangle \longrightarrow \langle o\!:\!Qu \,|\, Ev : \textsc{q}\rangle \; bind(o,m,Sig,In,C \;\#\; 0)$$

If m is defined locally in a class C with a matching signature, a process with the declared method code and local state is returned in a *bound* message. The object state is not upgraded at this point, so a match between the version numbers of C is not required for method binding. Otherwise, the *bind* message is retransmitted to the superclasses of C in a left-first depth-first order:

$$\begin{array}{l} bind(o,m,Sig,In,\varepsilon) \longrightarrow bound(o,none) \\ bind(o,m,Sig,In,(C \;\#\; n);\textsc{i}') \;\langle C \;\#\; n' : Cl \,|\, Inh : \textsc{i}, Mtds : \textsc{m}\rangle \\ \quad \longrightarrow \textbf{if}\; match(m,Sig,\textsc{m}) \;\textbf{then}\; bound(o,get(m,\textsc{m},In)) \;\textbf{else}\; bind(o,m,Sig,In,\textsc{i};\textsc{i}') \;\textbf{fi} \\ \qquad \langle C \;\#\; n\!:\!Cl \,|\, Inh : \textsc{i}, Mtds : \textsc{m}\rangle \end{array}$$

The auxiliary predicate $match(m,Sig,\textsc{m})$ is true if m is declared in $\textsc{m}$ with a signature Sig' such that $Sig \prec Sig'$, and the function get fetches method m in the method multiset $\textsc{m}$ of the class, and returns a process with the method's code and local state. Values of the actual in-parameters In, the caller o' and the label value n are stored in the local state. The resulting process w is loaded into the internal process queue of the callee, as defined by the rule:

$$bound(o,w) \;\langle o\!:\!Ob \,|\, PrQ : \textsc{w}\rangle \longrightarrow \langle o\!:\!Ob \,|\, PrQ : w;\textsc{w}\rangle$$

4.2 Upgrading Class Definitions

In order to control the upgrade propagation, class representations include a version number; i.e., a counter which records the number of times the class has been upgraded. Class upgrade may be direct or indirect through the upgrade of one of the superclasses. When a class is upgraded, its version number is incremented. Each time a (direct) superclass of a class C is upgraded, the version number of the class is incremented: although the definition of C itself has not changed, the class may have more attributes or methods by the way of inheritance. To propagate the upgrades properly, each class will record the version number of each of its inherited classes. Therefore the inherited classes are represented as a list of class names and version numbers:

$$\langle C \;\#\; n\!:\!Cl \,|\, Inh : (C_1 \;\#\; n_1);(C_2 \;\#\; n_2);\ldots, Att : \_, Mtds : \_, Tok : \_\rangle$$

Semantically a class upgrade is realized through the insertion of a new RL object $upgrade(C,\textsc{i},\textsc{a},\textsc{m})$ in the global state configuration at runtime, where C is the identifier of the class to be upgraded, $\textsc{i}$ is an inheritance list, $\textsc{a}$ is a state, and $\textsc{m}$ is a multiset of method definitions. The effect of the upgrade is that new inherited classes and attributes are added. Likewise new methods are added, but redefined methods must be treated differently: when a redefined method is added, the old version of the method must be removed. After the upgrade, the version number of the class is incremented. Let $\langle m, Sig, Body\rangle \in \textsc{m}$ denote that a method m with signature *Sig* and body *Body* is defined in a method multiset $\textsc{m}$. Define $\textsc{m} \oplus \textsc{m}'$ as $\textsc{m}' \cup \{\langle m, Sig, Body\rangle \mid \langle m, Sig, Body\rangle \in \textsc{m} \wedge \neg\exists Body'.\langle m, Sig, Body'\rangle \in \textsc{m}'\}$. The mechanism for *direct class upgrade* is captured in Creol's operational semantics by the following rule, which performs the upgrade using $\oplus$ to overwrite methods:

$$\begin{aligned}&\mathit{upgrade}(C, \textsc{i}', \textsc{a}', \textsc{m}')\langle C \ \#\ n : \mathit{Cl} \mid \mathit{Inh} : \textsc{i}, \mathit{Att} : \textsc{a}, \mathit{Mtds} : \textsc{m}, \mathit{Tok} : T\rangle\\ &\quad\longrightarrow \langle C \ \#\ (n+1) : \mathit{Cl} \mid \mathit{Inh} : \textsc{i}; \textsc{i}', \mathit{Att} : \textsc{a}; \textsc{a}', \mathit{Mtds} : \textsc{m} \oplus \textsc{m}', \mathit{Tok} : T\rangle\end{aligned}$$

When a class is upgraded by addition of some elements, its subclasses are also upgraded: although the definitions of the subclasses do not change, these classes indirectly acquire new attributes or methods by the way of inheritance. It is therefore necessary to propagate upgrade information to subclasses. The mechanism for *indirect class upgrade* is captured by the following equation:

$$\begin{aligned}&\langle C \ \#\ n : \mathit{Cl} \mid \mathit{Inh} : \textsc{i}; (C' \ \#\ n'); \textsc{i}'\rangle \ \langle C' \ \#\ n'' : \mathit{Cl} \mid \rangle\\ &\quad = \langle C \ \#\ (n+1) : \mathit{Cl} \mid \mathit{Inh} : \textsc{i}; (C' \ \#\ n''); \textsc{i}'\rangle \ \langle C' \ \#\ n'' : \mathit{Cl} \mid \rangle \ \textbf{\textit{if}} \ n'' > n'\end{aligned}$$

Note that the use of equations enables the version number update to execute in zero rewrite steps, which corresponds to locking the upgraded class object.

An example illustrates the effect of class upgrades at the semantic level. Let $\langle C \# 1 : \mathit{Cl} \mid \mathit{Inh} : (C_1 \ \#\ 1),\ \mathit{Att} : x; y; \textsc{a}, \ldots\rangle$ be the RL representation of a class C with parameters x and y, and attributes $\textsc{a}$. Let ε denote the empty list. The class will be upgraded with an additional ancestor class C_2 with an actual parameter x, by inserting the term $\mathit{upgrade}(C, (C_2(x) \ \#\ 1), \varepsilon, \varepsilon)$ into the global state of the running system. Later, the upgrade rule applies, resulting in the modified class representation $\langle C \ \#\ 2 : \mathit{Cl} \mid \mathit{Inh} : (C_1 \ \#\ 1); (C_2(x) \ \#\ 1), \mathit{Att} : x; y; \textsc{a}, \ldots\rangle$. If the class C_2 has been upgraded from its initial version, the equation reapplies, upgrading the version number of C_2 to the current version and incrementing the version number of the class C.

The example shows that we do not need knowledge of the actual version number of a class in the running system to add it as a superclass to the class we are upgrading, it suffices to use the initial version number. Consequently multiple upgrades do not cause upgrades to be forgotten, although the results of multiple (simultaneous) upgrades may vary due to the distributed topology reflected by the asynchronous upgrade rule.

The proposed class upgrade mechanism has some advantages. First, upgrade propagations are locally managed and classes need not know about their instances. Moreover, the version number recalls the number of changes applied to a class but old versions of a class are removed. Finally, there are no upgrade conflicts: one upgrade is performed at a time. If several upgrades redefine the same method the result may depend on the order in which the upgrades are performed, but the final result is stable. In order to enforce this discipline in a distributed setting where multiple copies of the class exist on different physical sites, there would typically be one master copy from which upgrades propagate to the other copies (the issue of duplicate classes is not treated here).

4.3 Upgrading Object Instances

In order to control the upgrades of object instances of an upgraded class, an object will include information about its current class version in its class attribute *Cl*. At initialization, the class attribute will store the name and current version of its class. When a class has been upgraded new object instances automatically get the new class attributes, due to the dynamic mechanism for collecting class variables (Sect. 4.1). However the upgrade of existing object instances of the class must be closely controlled.

Recall that the binding mechanism is dynamic: each time an object needs to evaluate a method, it requests the code associated with this method name. The code of suspended methods has already been loaded, and will be able to complete their evaluation. Problems may arise when calling new methods or new versions of methods using new attributes that are not presently available in the object. *The upgrade of an object has to be performed after the upgrade of its class and before new code which may rely on new class attributes is evaluated.* As processes may be recursive and even nonterminating, objects cannot generally be expected to reach a state without pending processes, even if the loading of processes corresponding to new method calls from the environment is postponed as in [1,9]. Consequently, it is too restrictive to wait for the completion of all pending methods before applying an upgrade. However, Creol objects may reach *quiescent* states when the processor has been released and before a pending process has been activated. In the case of process termination or an inner suspension point, *Pr* is empty. Any object which does not deadlock is guaranteed to eventually reach a quiescent state. In particular nonterminating activity is implemented by means of recursion, which ensures at least one quiescent state in each cycle. The mechanism for *object upgrade*, applied to quiescent states, is captured by the following equation:

$$\begin{aligned}&\langle o : Ob \mid Cl : C \,\#\, n, Pr : \varepsilon\rangle \langle C \,\#\, n' : Cl \mid Att : \mathrm{A}\rangle \\ &\quad = \langle o : Ob \mid Cl : C \,\#\, n', Pr : \varepsilon\rangle\ \langle C \,\#\, n' : Cl \mid Att : \mathrm{A}\rangle\ getAttr(o, C, \mathrm{A})\ \textbf{\textit{if}}\ n' > n\end{aligned}$$

A similar equation handles local synchronous calls.

Due to the implicit inheritance graph, upgrade of attributes is handled as standard object instantiation; this is given by the equations for *getAttr*, which recursively compute an object state from provided values for class parameters. For object upgrade the present object state A replaces the initial values, thus only new attributes get values computed while inspecting the inheritance graph starting at class *C*. The use of equations corresponds to locking the object. Evaluation results in a term $gotAttr(o, \mathrm{A}')$ where A' is the resolved attribute list with values. The mechanism for *state upgrade*, replacing the old object state by the the new one, is captured by the following equation:

$$gotAttr(o, \mathrm{A}')\ \langle o : Ob \mid Att : \mathrm{A}\rangle = \langle o : Ob \mid Att : \mathrm{A}'\rangle$$

The described runtime mechanism allows the upgrade of active objects. Attributes are collected at upgrade time while code is loaded “on demand”. A class may be upgraded several times before the object reaches a quiescent state, so the object may have missed some upgrades. However a single state upgrade suffices to ensure that the object, once upgraded, is a complete instance of the present version of its class. The upgrade mechanism ensures that an object upgrade has occurred before new code is evaluated.

After an upgrade the object is in a transitional mode, where its attributes are new but old code may still occur in the process queue. This explains why attributes may neither be removed nor change types during class upgrade. With this restriction, the evaluation of old code can be completed without errors. Notice that if a call to a redefined method *m* appears in remaining old code, the call will nevertheless be bound to the new version of *m*. This does not cause difficulties provided that the restrictions to covariance for in-parameters and contravariance for out-parameters are respected. This way, the use of recursion rather than while-loops allows a smooth upgrade of nonterminating activity.

4.4 Example: Analysis of a Bank Account Upgrade

The bank account example of Sect. 3.1 is now reconsidered to illustrate the operational semantics and its executable aspect, in order to provide some insight into the behavior of the asynchronous update mechanism. We define an initial configuration consisting of the original bank class and two new bank accounts *b* and *c* together with a deposit invocation, say $!b.deposit(100)$, followed by a transfer $!b.transfer(200,c)$. The transfer will be suspended since the balance is not large enough. We then augment the initial configuration with the class upgrade

```
upgrade(BankAccount, ε, overdraft : Nat =100,
   op transfer (in sum : Nat, acc : Account) ==
     await bal ≥ sum−overdraft; bal := bal−sum; acc.deposit(sum)  )
```

In order to see the effect of executing the operational semantics, we use Maude's search facilities to search for all possible final states. The search results in two solutions: Both have succeeded in upgrading the bank account objects. In one solution the *b* account has a final balance of 100 and a pending transfer invocation which cannot be completed, whereas the other solution has a final balance of -100 and no pending code. In the the first solution the transfer invocation is bound before bank account *b* is upgraded, with the result that the transfer is suspended and cannot be completed (since it awaits $bal \geq sum$). In the second solution, the bank account is upgraded before the transfer invocation is bound, with the result that the transfer is completed (since the upgraded transfer awaits $bal \geq sum - overdraft$).

Can we guarantee that an upgrade will succeed? In order to illustrate this problem, we introduce a nonterminating activity: Let an object recursively make asynchronous calls $!b.deposit(0)$ (which have no effect on the state of *b*). In this case a search for final states does not succeed, but we can search for all solutions for *N* deposit calls. Ignoring pending calls, there are two solutions for every fixed *N*: one solution has an upgraded class and the other does not. The analysis suggests that there is a race condition between the evaluation of *bind* and *update* messages with regard to the class representation in the global configuration. As the update rule is continuously enabled, weak fairness is needed to guarantee that the update will succeed. For simulation Maude's predefined fair rewrite strategy ensures that class updates will eventually be applied. In contrast the update rule for the object state is only enabled in quiescent states. Unless the object deadlocks quiescent states occur regularly, which suggests that strong fairness is needed to ensure that the update is applied. This problem is circumvented by using equations in RL; state updates have priority and will always be selected if enabled.

5 Related Work

Although many approaches to reconfigurable distributed systems [2, 4, 6, 14, 16, 21, 26] do not address availability requirements during reconfiguration, availability is an essential feature of many modern distributed applications. Dynamic or online system upgrade considers how running systems may evolve. Recently, several authors have investigated type-safe mechanisms for runtime upgrade of imperative [29], functional [5], and object-oriented [10] languages. The latter paper considers object instance evolution (reclassification) in Fickle, based on a type system which guarantees type safety when an object changes its class. These approaches consider the upgrade of single type declarations, procedures, objects, or components in the sequential setting. Fickle has been extended to multithreading [9], but restrictions to runtime reclassification are needed; e.g., an object with a nonterminating (recursive) method will not be reclassified.

Work on version control for modular systems aims at more generic upgrade support. Some approaches allow multiple versions of a module to coexist after an upgrade [3, 5, 11, 13, 15], while others keep only the latest version by performing a global update or "hot-swapping" [1, 7, 22, 25]. Another important distinction between different approaches is their treatment of active behavior. Upgrade of active behavior may be disallowed [7, 13, 22, 25], delayed [1, 9], or supported [15, 29]. Most approaches favoring global updates do not support the upgrade of an active module running the old version. A system which addresses the upgrade of active code is proposed in [29] for the setting of type declarations and procedures in (sequential) C. However, the approach is synchronous in the sense that upgrades which cannot be applied immediately will fail.

Dynamic class constructs may be considered as a form of version upgrade. Hjálmtýsson and Gray [15] propose an approach for C++ based on proxy classes through which the actual class is linked (reference indirection). Their approach supports multiple versions of each class. Because existing objects of the class are not upgraded, activity in existing objects is uninterrupted. Dynamic class upgrade in Java has been proposed using proxy classes [25] and by modifying the Java virtual machine [22]. Both approaches are based on global upgrade, but the approaches are not applicable to active objects.

Automatic upgrade based on lazy global update is addressed in [1] for distributed objects and in [7] for persistent object stores. Although the object instances of upgraded classes are upgraded in these works, inheritance is not addressed which limits the effect of class upgrade. Further, these approaches cannot handle (nonterminating) active code. Our approach supports multiple inheritance, but restricts upgrades to addition and redefinition and may therefore avoid these limitations. Only one version of an upgraded class is kept in the system but active objects may still be upgraded. Upgrade is asynchronous and distributed, and may therefore be temporarily delayed.

6 Conclusion

Many critical distributed systems need to be modified without compromising availability requirements. This paper exploits the class structure of object-oriented programs to introduce evolution of the inheritance graph at runtime. We have presented a novel construct for dynamic class upgrade in distributed object-oriented systems and formalized its operational semantics in rewriting logic. Upgrading a class has an effect on all its subclasses and all object instances of these classes. The construct allows classes to be extended with new attributes, new methods, and new ancestor classes, while existing methods may be redefined. A subtype relationship is needed for the redefinition of methods, while extension is not restricted. Active and nonterminating code may be upgraded. The mechanism ensures that virtual binding will still succeed after an upgrade.

Our formalization uses equations to update class version numbers for indirect class upgrade and to upgrade individual objects. This seems natural at a high level of abstraction. At a lower level of abstraction this semantics may lead to temporary locks on objects, since equations apply between rewrite steps. It is therefore of interest to investigate how these equations may be replaced by rules. In particular the equations for object upgrade could be reformulated as rules. However this would require the messages controlling method binding and attribute updating to include version number information, using conditional rules to ensure consistent version numbering.

In future work, we plan to study how dynamic class constructs as proposed in this paper may be improved through type analysis and provide formal proof that such upgrade mechanisms preserve strong typing. Furthermore it is interesting to consider upgrade mechanisms addressing several (hierarchies of) classes simultaneously. Such mechanisms could probably allow a more flexible notion of upgrade. In particular mutual and cyclic dependencies between objects could be addressed directly in the same upgrade. It seems probable that such package upgrades may require a more synchronized upgrade mechanism than the mechanism proposed here, resulting in considerably more overhead in the distributed concurrent setting.

References

1. S. Ajmani, B. Liskov, and L. Shrira. Scheduling and simulation: How to upgrade distributed systems. In *9th Workshop on Hot Topics in Operating Systems (HotOS-IX)*, pages 43–48. USENIX, May 2003.
2. J. P. A. Almeida, M. Wegdam, M. van Sinderen, and L. Nieuwenhuis. Transparent dynamic reconfiguration for CORBA, *Proc. 3rd Intl. Symp. on Distributed Objects and Applications (DOA)*, pages 197–207. IEEE CS Press, Sep. 2001.
3. J. L. Armstrong and S. R. Virding. Erlang - an experimental telephony programming language. In *XIII Intl. Switching Symposium*, June 1990.
4. C. Bidan, V. Issarny, T. Saridakis, and A. Zarras. A dynamic reconfiguration service for CORBA. In *4th Intl. Conf. on Configurable Dist. Systems*, pages 35–42. IEEE, May 1998.
5. G. Bierman, M. Hicks, P. Sewell, and G. Stoyle. Formalizing dynamic software updating. In *Proc. of the 2nd Intl. Workshop on Unanticipated Software Evolution (USE)*, Apr. 2003.
6. T. Bloom. *Dynamic Module Replacement in a Distributed Programming System*. PhD thesis, MIT, 1983. Also available as MIT LCS Tech. Report 303.
7. C. Boyapati, B. Liskov, L. Shrira, C.-H. Moh, and S. Richman. Lazy modular upgrades in persistent object stores. In R. Crocker and G. L. S. Jr., editors, *Proc. ACM Conf. on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'03)*, pages 403–417. ACM Press, 2003.
8. 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.
9. F. Damiani, M. Dezani-Ciancaglini, and P. Giannini. Re-classification and multithreading: Fickle$_{\mathrm{MT}}$. In *Proc. Symp. on Applied Computing (SAC'04)*, pages 1297–1304. ACM, 2004.

10. S. Drossopoulou, F. Damiani, M. Dezani-Ciancaglini, and P. Giannini. More dynamic object re-classification: Fickle$_{\text{II}}$. *ACM Trans. on Prog. Lang. and Systems*, 24(2):153–191, 2002.
11. D. Duggan. Type-Based hot swapping of running modules. In C. Norris and J. J. B. Fenwick, editors, *Proc. of the 6th Intl. Conf. on Functional Programming(ICFP 01)*, volume 36, 10 of *ACM SIGPLAN notices*, pages 62–73, New York, Sept. 2001. ACM Press.
12. J. A. Goguen, T. Winkler, J. Meseguer, K. Futatsugi, and J.-P. Jouannaud. Introducing OBJ. In J. A. Goguen and G. Malcolm, editors, *Software Engineering with OBJ: Algebraic Specification in Action*, Advances in Formal Methods, chapter 1, pages 3–167. Klüwer, 2000.
13. D. Gupta, P. Jalote, and G. Barua. A formal framework for on-line software version change. *IEEE Trans. Software Eng.*, 22(2):120–131, 1996.
14. R. S. Hall, D. Heimbigner, A. van der Hoek, and A. L. Wolf. An architecture for post-development configuration management in a wide-area network. In *Intl. Conf. on Dist. Computing Systems*, pages 269–279. IEEE CS Press, May 1997.
15. G. Hjálmtýsson and R. S. Gray. Dynamic C++ classes: A lightweight mechanism to update code in a running program. In *Proc. 1998 USENIX Technical Conf.* USENIX, May 1998.
16. C. R. Hofmeister and J. M. Purtilo. A framework for dynamic reconfiguration of distributed programs. Technical Report CS-TR-3119, University of Maryland, College Park, 1993.
17. E. B. Johnsen and O. Owe. An asynchronous communication model for distributed concurrent objects. In *Proc. 2nd IEEE Intl. Conf. on Software Engineering and Formal Methods (SEFM'04)*, pages 188–197. IEEE CS Press, Sept. 2004.
18. E. B. Johnsen and O. Owe. Inheritance in the presence of asynchronous method calls. In *Proc. 38th Hawaii Intl. Conf. on System Sciences (HICSS'05)*. IEEE CS Press, Jan. 2005.
19. E. B. Johnsen, O. Owe, and E. W. Axelsen. A run-time environment for concurrent objects with asynchronous methods calls. In *Proc. 5th Intl. Workshop on Rewriting Logic and its Applications (WRLA'04)*, Electr. Notes Theor. Comput. Sci. 117: 375–392, Jan. 2005.
20. A. Ketfi and N. Belkhatir. A metamodel-based approach for the dynamic reconfiguration of component-based software. In J. Bosch and C. Krueger, editors, *Proc. Intl. Conf. on Software Reuse 2004*, LNCS 3107, pages 264–273. Springer, 2004.
21. J. Kramer and J. Magee. The Evolving Philosophers Problem: Dynamic change management. *IEEE Trans. Software Eng.*, 16(11):1293–1306, Nov. 1990.
22. S. Malabarba, R. Pandey, J. Gragg, E. Barr, and J. F. Barnes. Runtime support for type-safe dynamic Java classes. In E. Bertino, editor, *14th European Conf. on Object-Oriented Programming (ECOOP'00)*, LNCS 1850, pages 337–361. Springer, June 2000.
23. J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96:73–155, 1992.
24. E. Najm and J.-B. Stefani. A formal semantics for the ODP computational model. *Computer Networks and ISDN Systems*, 27:1305–1329, 1995.
25. A. Orso, A. Rao, and M. J. Harrold. A technique for dynamic updating of Java software. In *Proc. Intl. Conf. on Software Maintenance (ICSM 2002)*, pages 649–658. IEEE CS Press, Oct. 2002.
26. T. Ritzau and J. Andersson. Dynamic deployment of Java applications. In *Java for Embedded Systems Workshop*, London, May 2000.
27. C. A. N. Soules, *et al*. System support for online reconfiguration. In *Proc. 2003 USENIX Technical Conf.*, pages 141–154. USENIX, 2003.
28. I. Stojmenović, M. Seddigh, and J. Zunic. Dominating sets and neighbor elimination based broadcasting algorithms in wireless networks. *IEEE Trans. on Parallel and Distributed Systems*, 13:14–25, 2002.
29. G. Stoyle, M. Hicks, G. Bierman, P. Sewell, and I. Neamtiu. *Mutatis Mutandis*: Safe and flexible dynamic software updating. In *Proc. of the Conf. on Principles of Programming Languages (POPL'05)*, pages 183–194. ACM Press, Jan. 2005.
30. A. Yonezawa. *ABCL: An Object-Oriented Concurrent System*. The MIT Press, 1990.