Type-Safe Runtime Class Upgrades in Creol

Ingrid Chieh Yu, Einar Broch Johnsen, and Olaf Owe

Department of Informatics, University of Oslo
PO Box 1080 Blindern, NO-0316 Oslo, Norway
{ingridcy,einarj,olaf}@ifi.uio.no

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 update, as encapsulation clearly separates internal structure and external services. This paper considers a type-safe asynchronous mechanism for dynamic class upgrade, allowing class hierarchies to be updated in such a way that the existing objects of the upgraded class and of its subclasses gradually evolve at runtime. New external services may be introduced in classes and old services may be reprogrammed while static type checking ensures that asynchronous class updates maintain type safety. A formalization is shown in the Creol language which, addressing distributed and object-oriented systems, provides a natural framework for dynamic upgrades.

1 Introduction

Long-lived distributed applications with high availability requirements need the ability to adapt to new requirements that arise over time without compromising application availability. These requirements include bugfixes but also new or improved features. Examples of such applications are found in financial transaction processes, aeronautics and space missions, and mobile and Internet applications. In these examples, updates must be applied at runtime. Early approaches to software updates [4, 12, 16] do not address the issue of continuous availability, but runtime reconfiguration and upgrade have recently attracted attention [1–3, 5, 9–11, 17, 19, 21]. In large distributed systems runtime updates need to be applied in an asynchronous and modular way, and propagate gradually through the distributed system. An appropriate update system should [1, 21]: propagate updates automatically, provide a means to control *when* components may be upgraded, and ensure the availability of system services during the upgrade process.

This paper considers a type-safe mechanism for distributed runtime updates in Creol [13], a formally defined object-oriented language which specifically targets open distributed systems. We consider updates in the form of runtime upgrades of existing classes combined with runtime additions of new interfaces and new classes. Upgrading a class affects all future and existing object instances of the class and its subclasses. As runtime upgrades are handled by asynchronous messages, allowing message overtaking, dependencies between different upgrades

In: R. Gorrieri and H. Wehrheim (Eds.):
Proc. 8th Intl. Conf. on Formal Methods for Open Object-Based Distributed Systems
(FMOODS'06), p. 202–217, LNCS 4037, Springer-Verlag, June 2006.

could violate type safety. Extending previous work [14], this paper introduces a type system for class upgrades which derives the upgrade dependencies of each upgrade. These dependencies enforce an ordering of the upgrades in the runtime system, formalized in rewriting logic [18], which ensures that the application of the distributed upgrades is type-safe. Consequently, runtime class upgrades will not introduce type errors. The upgrade mechanism proposed in this paper allows new interfaces to be added to classes at runtime. This way upgraded classes may provide new external services. The following simple example illustrates dependencies between several updates.

Motivating example. We adopt a separation of concerns between external service specifications, given as interfaces, and implementation code, organized in classes. Object pointers are typed by interfaces while objects are instances of classes. A type system is used to ensure that methods invoked on object pointers are supported by the objects. Consider a simple scenario with three classes C_1, C_2, and C_3, where C_3 inherits C_2 (the comment *V:1* means version 1 of a class):

```
class C1 --- V:1, U:0      class C2 --- V:1, U:0      class C3 --- V:1, U:0
begin                      begin   end                inherits C2
op run() == n(); run()                                begin   end
op n() == skip
end
```

The example sketch is given in Creol, *U:0* comments that a class has not (yet) been upgraded. Here, C_1 objects are active as the *run* method is activated at object creation, with a nonterminating behavior consisting of repeated local calls to a method n. The external functionality of each class is given by its interfaces. None are given here, so in this example only internal calls are possible in C_1.

By *dynamically upgrading* the class C_2 with a new method m, this method will become available via objects of classes C_2 and its subclass C_3. However, after the update the new method is only known internally in these classes. In order to *export* the new functionality, we dynamically add a new interface I providing a method m with an appropriate signature, after which m may be invoked on pointers typed by I. If we can type check that C_3 implements I, it is type-safe to bind a pointer typed by I to an instance of C_3 and invoke the new method m on this object. This may be achieved by dynamically redefining method n in class C_1 to create an appropriately typed instance of C_3 and invoke m on this instance, for instance by the code $\mathbf{var}\ x : I; x := \mathbf{new}\ C_3(); x.m()$. These dynamic updates may be realized by four update messages added to the running system: introducing I, upgrading C_1 by the redefinition of n, C_2 by a new method m, and C_3 by the new interface I. After successful upgrades (*U:1*), the following classes replace the previous runtime class definitions:

```
class C1 --- V:2, U:1      class C2 --- V:2, U:1      class C3 --- V:3, U:1
begin                      begin                      implements I
op run() == n(); run()     op m() == Body             inherits C2
op n() == var x : I;       end                        begin   end
   x := new C3(); x.m()
end
```

Furthermore, the active behavior of existing instances of C_1 now create instances of C_3 on which the new method m is invoked.

A type-safe introduction of these upgrades in a distributed system requires a combination of type checking and careful timing at runtime. In particular, the redefinition of method n has an immediate effect on any instance of C_1. In order to avoid errors, this upgrade cannot be applied *before* C_3 implements the new interface I. However, the addition of the new interface requires the presence of method m, which in turn requires that the application of the upgrade of C_2 has *already* occurred. In fact, C_3 has been upgraded twice, once directly and once indirectly through the upgrade of C_2. This paper formalizes an asynchronous update mechanism which handles these dependencies, maintaining runtime type safety throughout the upgrade process.

Paper overview. Sect. 2 introduces behavioral interfaces, Sect. 3 summarizes Creol, Sect. 4 presents Creol's type system, and Sect. 5 presents the dynamic class construct. Sect. 6 discusses related work and Sect. 7 concludes the paper.

2 Behavioral Interfaces

An object may assume different roles, depending on the context of interaction, which are captured by specifications of aspects of its externally observable behavior. A *behavioral interface* consists of a set of method names with signatures and semantic constraints on the use of these methods. In this paper we restrict semantic constraints to cointerface requirements, explained as follows: For active objects it may be desirable to restrict access to the methods in an interface to calling objects of a particular *cointerface.* This way the called object may invoke methods of the caller and not only passively complete invocations of its own methods, thus providing support for callback. *Mutual dependency* is specified if two interfaces have each other as cointerface. Let *Any* be the superinterface of all interfaces; *Any* is used as cointerface if no callback knowledge is required.

Object references (pointers) are typed by behavioral interfaces. References typed by different interfaces may refer to the same object identifier. A class *implements* an interface if its object instances provide the behavior described by the interface. A class may implement several interfaces and different classes may implement the same interface. Reasoning control is ensured by interface-level substitutability: *a reference typed by an interface I may be replaced by another reference typed by I or by a subinterface of I.* This substitutability is reflected in the executable language by the fact that late binding applies to all external method calls, as the runtime class of the object need not be statically known.

Let τ_B be a set of basic data type names and $\tau_{\mathcal{I}}$ a set of interface names, such that $\tau_B \cap \tau_{\mathcal{I}} = \emptyset$. Let τ denote the set of all types; $\tau_B \subseteq \tau$ and $\tau_{\mathcal{I}} \subseteq \tau$. Let I and J be typical elements of $\tau_{\mathcal{I}}$, and T of τ. We assume that τ_B includes standard types such as Booleans and natural numbers. Type schemes such as parametrized data types may be applied to types in τ to form new types in τ, $\mathsf{Set}[T]$ and $\mathsf{List}[T]$ are included among the type schemes. To conveniently organize object viewpoints, interfaces may be structured in an inheritance hierarchy.

Definition 1. *An* interface *is denoted by a term $\mathit{int}\,(\mathit{Inh}, \mathit{Mtd})$ of type $\mathcal{I}$, where Inh is a list of (inherited) interfaces and Mtd is a set of method declarations $\mathit{mdecl}\,(\mathit{Nm}, \mathit{Co}, \mathit{In}, \mathit{Out})$, where Nm is a method name, Co is a cointerface, and In and Out are lists of parameter types.*

Dot notation is used to access the elements of tuples such as methods and interfaces; e.g., $\mathit{int}\,(\mathit{Is}, M).\mathit{Mtd} = M$. The empty list is denoted ε. The name $\mathit{Any} \in \tau_{\mathcal{I}}$ is reserved for $\mathit{int}\,(\varepsilon, \emptyset)$, and the name $\mathit{Internal} \in \tau_{\mathcal{I}}$ is reserved for type checking purposes (see Sect. 3). If I inherits J, the methods of both I and J must be available in any class that implements I. We consider a nominal subtype relation [20] for interfaces. Two interfaces with the same set of methods may be part of different subtype relationships.

3 Creol: A Language for Distributed Concurrent Objects

Creol is a high-level object-oriented language targeting open distributed systems by combining interface types and concurrent objects with asynchronous method calls, and by combining active and reactive object behavior [13,15]. In this paper blocking and nonblocking (suspending) method calls are considered, although the results of the paper apply to the full language. An object has its own processor which evaluates local processes. Processes result from method activations. Active behavior is initiated by the special *run* method, activated at object creation, and interleaved with reactive behavior by means of suspension. Due to suspension, the values of object variables may depend on the nondeterministic interleaving of processes, so local process variables supplement the object variables and include the formal parameters. An object may contain several (pending) activations of a method, possibly with different values for local variables.

Objects only interact through asynchronous method calls. 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 execution of the method activations in another order. A *blocking* call $x.m(\text{E}; \text{V})$ immediately blocks the processor while waiting for a reply. A *nonblocking* call **await** $x.m(\text{E}; \text{V})$ releases the processor while waiting for a reply, allowing other processes to execute. When the reply arrives, the suspended process becomes enabled and evaluation may resume. This approach provides flexibility in the distributed setting: suspended processes or new method activations 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. After processor release, any enabled pending process may be selected for evaluation. When x evaluates to *self*, the call is said to be local. *Internal* calls are not prefixed by an object identifier and are identified syntactically, otherwise the call is external. All internal calls are here late bound.

The language distinguishes data, typed by data types, and objects, typed by interfaces. We assume given a *strongly typed functional language* of well-typed expressions $e \in \mathsf{Expr}$ without side effects, including two subtypes $\mathsf{ObjExpr}$ and

$$\begin{array}{lll}
\mathit{CL} & ::= & [\textbf{class}\ C\ [(\mathit{Vdecl})]^{?}\ [\textbf{implements}\ [I]^{+}_{;}]^{?}\ [\textbf{inherits}\ [C[(\textsc{e})]^{?}]^{+}_{;}]^{?} \\
 & & \quad \textbf{begin}\ [\textbf{var}\ \mathit{Vdecl}]^{?}\ [[\textbf{with}\ I]^{?}\ \mathit{Methods}]^{*}\ \textbf{end}]^{*} \\
\mathit{Methods} & ::= & [\textbf{op}\ m\ ([\textbf{in}\ \mathit{Vdecl}]^{?}\ [\textbf{out}\ \mathit{Vdecl}]^{?}) == [\textbf{var}\ \mathit{Vdecl};]^{?}\ \textsc{s}]^{+} \\
\mathit{Vdecl} & ::= & [v : T]^{+}_{;}
\end{array}$$

Figure 1. An outline of the language syntax for classes, excluding expressions e, expression lists $\textsc{e}$, and statement lists $\textsc{s}$. The meta notation $[\ldots]^{?}$ denotes optional parts, $[\ldots]^{*}$ repetition zero or more times, and $[\ldots]^{+}_{d}$ non-empty repetition with d as delimiter.

BoolExpr whose expressions reduce to object references (typed by interface) and Booleans, respectively. There are no constructors or field access functions for terms in ObjExpr, but variables bound to object references may be compared by an equality function. Let Γ_F be a typing environment which includes all relevant type information for the constants and functions of the functional language, and let Γ extend Γ_F with variable declarations. Then $\Gamma \vdash_{\mathrm{F}} e : T$ denotes that e has type T in Γ. It is assumed that expressions are *type-sound*: well-typed expressions remain well-typed during evaluation. If $\Gamma \vdash_{\mathrm{F}} e : T$ and e reduces to e', then $\Gamma \vdash_{\mathrm{F}} e' : T'$ such that $T' \preceq T$.

Object-oriented features extend the functional language. Class definitions include declarations of persistent state variables and method definitions.

Definition 2. *A* class *is denoted by a term* class$(\mathit{Par}, \mathit{Upg}, \mathit{Imp}, \mathit{Inh}, \mathit{Var}, \mathit{Mtd})$*, where Par is a list of typed program variables, Upg the current upgrade number, Imp a list of interface names, Inh a list of class names, defining class inheritance, Var a list of typed program variables (possibly with initial expressions), and Mtd a set of methods* $\mathit{mtd}(\mathit{Nm}, \mathit{Co}, \mathit{In}, \mathit{Out}, \mathit{Body})$ *where Nm is a method name, Co an interface, In and Out lists of variable declarations, and Body a pair of variable declarations Vdecl and statements* $\textsc{s}$*.*

The *Upg* attribute is not a part of the Creol syntax and cannot be altered by programmers. For internal methods, the cointerface field is *Internal*. The field *Imp* represents interfaces supported by this class. The typing of remote method calls in a class C relies on the fact that the calling object supports the interfaces of C, and these are used to check any cointerface requirements of the calls.

Let τ_C denote the set of class names, with typical element C, and $\mathcal{C}$ the set of class terms. An abstract representation of a class may be given following the BNF syntax of Figure 1. Method declarations in classes consist of local variable declarations and a list of program statements (see Figure 2). Assignment to local and object variables is expressed as $\textsc{v} := \textsc{e}$ for a disjoint list of program variables $\textsc{v}$ and an expression list $\textsc{e}$, of matching types. In-parameters as well as the pseudo-variables *self*, for self reference, and *caller* are read-only variables.

Due to the interface typing of object variables, the actual class of the receiver of an external call is not statically known. Consequently, external calls are *late bound*. Let the nominal subtype relation $\preceq$ be a reflexive partial ordering on types, including interfaces. The nominal subtype relation restricts a structural subtype relation which ensures substitutability; If $T \preceq T'$ then any value of T may masquerade as a value of T' [20]. For product types R and R', $R \preceq R'$

Syntactic categories.		*Definitions.*
s in Stm	v in Var	$p ::= m \mid x.m$
m in Mtd	p in $\mathsf{MtdCall}$	$\mathrm{S} ::= s \mid s; \mathrm{S}$
e in Expr	x in $\mathsf{ObjExpr}$	$s ::= \textbf{skip} \mid \mathrm{V} := \mathrm{E} \mid v := \textbf{new}\ C(\mathrm{E}) \mid p(\mathrm{E}; \mathrm{V}) \mid \textbf{await}\ p(\mathrm{E}; \mathrm{V})$

Figure 2. Program statements in method definitions, with typical terms for each category. Capitalized terms such as E denote lists of the given syntactic categories.

is the point-wise extension of the subtype relation. To explain the typing and binding of methods, $\preceq$ is extended to function spaces $A \to B$, where A and B are (possibly empty) product types: $A \to B \preceq A' \to B' \Leftrightarrow A' \preceq A \wedge B \preceq B'$. The static analysis of an internal call $m(\mathrm{E}; \mathrm{V})$ or **await** $m(\mathrm{E}; \mathrm{V})$ will assign unique types to the in- and out-parameters depending on the textual context, say $\mathrm{E} : T_{\mathrm{E}}$ and $\mathrm{V} : T_{\mathrm{V}}$. The call is *type-correct* if there is a method declaration $m : T_1 \to T_2$ in the class C such that $T_1 \to T_2 \preceq T_{\mathrm{E}} \to T_{\mathrm{V}}$. An external call $o.m(\mathrm{E}; \mathrm{V})$ or **await** $o.m(\mathrm{E}; \mathrm{V})$ to an object o of interface I is type-correct if it can be bound to a method declaration in I in a similar way. The static analysis of a class will verify that it implements its declared interfaces. Assuming that any object variable typed by I is an instance of a class implementing I, method binding at runtime will succeed regardless of the dynamically identified class of the object.

4 Typing

The typing environment Γ in Creol's nominal type system is a *mapping family*: $\Gamma_{\mathcal{I}}$ maps interface names to interfaces, $\Gamma_{\mathcal{C}}$ class names to classes, and Γ_{V} program variable names to types. Without class upgrades, $\Gamma_{\mathcal{I}}$ and $\Gamma_{\mathcal{C}}$ correspond to static tables. Declarations may only update Γ_{V}, and program statements may not update Γ_{V}. For the purposes of dynamic updates, a *dependency mapping* Γ_d captures the dependencies that a class has to different classes in the program.

Definition 3. *The* dependency mapping $\Gamma_d : \tau_{\mathcal{C}} \times \mathsf{Nat} \to \mathsf{Set}[\tau_{\mathcal{C}} \times \mathsf{Nat}]$ *maps pairs of class names and upgrade numbers to sets of such pairs.*

Each upgrade of a class C is uniquely identified by a pair $\langle C, u\rangle$. Thus, elements in $\Gamma_d(\langle C, u\rangle)$ represent classes on which upgrade u of class C depends, and structural requirements to these classes. At runtime Γ_d helps to monitor whether these structural requirements are fulfilled, and to enforce an ordering of local updates obeying the dependency requirements.

The type analysis of a syntactic construct D is formalized by a deductive system for judgments $\Gamma \vdash D\,\langle\Delta\rangle$, where Γ is the typing environment and Δ the *update* of the typing environment. After analysis of D, the typing environment becomes Γ *overridden by* Δ, denoted $\Gamma + \Delta$. Sequential composition has the rule

$$\text{(SEQ)}\quad \frac{\Gamma \vdash D\,\langle\Delta\rangle \qquad \Gamma + \Delta \vdash D'\,\langle\Delta'\rangle}{\Gamma \vdash D; D'\,\langle\Delta + \Delta'\rangle}$$

where $+$ is an associative operator on mappings with the identity element $\emptyset$. We abbreviate $\Gamma \vdash D\,\langle\emptyset\rangle$ to $\Gamma \vdash D$. Mapping families are now formally defined.

Definition 4. *Let n be a name, d a declaration, $i \in I$ a mapping index, and $[n \mapsto_i d]$ the binding of n to d indexed by i. A* mapping family *Γ is built from the empty mapping family $\emptyset$ and indexed bindings by the constructor $+$. The* extraction *of an indexed mapping Γ_i from Γ and* application *for the indexed mapping Γ_i, are defined as follows*

$$\begin{aligned}
&\emptyset_i &&= \varepsilon \\
&(\Gamma + [n \mapsto_{i'} d])_i &&= \mathbf{if}\ i = i'\ \mathbf{then}\ \Gamma_i + [n \mapsto_i d]\ \mathbf{else}\ \Gamma_i \\
&\varepsilon(n) &&= \bot \\
&(\Gamma_i + [n \mapsto_i d])(n') &&= \mathbf{if}\ n = n'\ \mathbf{then}\ d\ \mathbf{else}\ \Gamma_i(n').
\end{aligned}$$

A class or interface declaration binds a name to a class or interface term, respectively. Class and interface names need not be distinct. A program consists of a list of interface and class declarations, represented by the mappings $\Gamma_{\mathcal{I}}$ and $\Gamma_{\mathcal{C}}$. For type checking a program, each interface and class term is type checked based on these mappings (binding *self* to the class name in the second case). The type rules are given in Figure 3 (omitting the rule for interfaces). To simplify the exposition, some auxiliary functions are used to retrieve information from the typing environment. The predicate *matchpar* verifies that the formal and actual parameters of (inherited) classes match, given a list of classes and a typing environment. The predicate *matchext* checks that an external invocation may be bound through the interface of the callee, based on the types of actual parameter values and the possible cointerfaces of the caller. The function *matchint* returns a list of classes in which an internal invocation may be bound given a method, a list of classes, and a typing environment. This function is used to check that a class provides method bodies for the method declarations of its interfaces, and for type checking internal calls. The function *InhAttr* returns a list of typed variables when given a list of classes and a typing environment, and is used to extend the typing environment with inherited attributes.

The main type rules are now briefly explained. Programs are type checked in the context of Γ_F. Variable declarations extend the context used to type check methods in rule (Class). Local variable declarations extend the typing environment used to type check the program statements of a method in rule (method). For object creation, (new) ensures that the class must implement an interface which is a subtype of the declared interface of the object pointer. For external calls $x.m$, (ext) checks that the interface of x offers a method m with a cointerface implemented by the class of the caller. Consequently, *remote calls* to *self* are allowed when the class implements an interface used as the cointerface of the method in the current class. For internal calls m, (int) checks that the method has cointerface *Internal*. For a variable occurring in a method body, the pair consisting of the name of the class in which the variable is declared and the upgrade number of this class, are added to the dependency mapping for the method. Similarly, matching classes for internal calls and object creations also extend the mapping. This way, the type system constructs a dependency mapping which captures the dependencies a method has to different classes in the program. This dependency mapping is exploited for system upgrades.

$$\text{(PROG)}\ \frac{\forall I \in \tau_{\mathcal{I}} \cdot \varGamma_{\mathcal{I}} \vdash \varGamma_{\mathcal{I}}(I) \quad \forall C \in \tau_{\mathcal{C}} \cdot \varGamma_F + \varGamma_{\mathcal{I}} + \varGamma_{\mathcal{C}} + [\mathit{self} \mapsto_{\textsc{v}} C] \vdash \varGamma_{\mathcal{C}}(C)}{\varGamma_F \vdash \varGamma_{\mathcal{I}}, \varGamma_{\mathcal{C}}}$$

$$\text{(CLASS)}\ \frac{\begin{array}{c}\varGamma \vdash \mathit{Par}\,\langle \varDelta \rangle \qquad \varGamma + \varDelta \vdash \mathit{InhAttr}(\mathit{Inh}, \varGamma_{\mathcal{C}}), \mathit{Var}\,\langle \varDelta' \rangle \\ \mathit{matchpar}(\varGamma + \varDelta, \mathit{Inh}) \qquad \forall m \in \mathit{Mtd} \ \cdot \ \varGamma + \varDelta + \varDelta' \vdash m\,\langle \varDelta^m \rangle \\ \forall I \in \mathit{Imp} \cdot \forall m \in \varGamma_{\mathcal{I}}(I).\mathit{Mtd} \cdot \mathit{matchint}(m, \varGamma_{\textsc{v}}(\mathit{self}), \varGamma) \neq \varepsilon\end{array}}{\varGamma \vdash \mathit{class}\,(\mathit{Par}, \mathit{Upg}, \mathit{Imp}, \mathit{Inh}, \mathit{Var}, \mathit{Mtd})\,\langle \varDelta + \varDelta' + \bigcup_{m \in \mathit{Mtd}} \varDelta^m \rangle}$$

$$\text{(METHOD)}\ \frac{\varGamma \vdash (\mathit{caller} : \mathit{Co}); \mathit{In}; \mathit{Out}; \mathit{Body}\,\langle \varDelta \rangle}{\varGamma \vdash \mathit{mtd}\,(\mathit{Nm}, \mathit{Co}, \mathit{In}, \mathit{Out}, \mathit{Body})\,\langle \varDelta_d \rangle}$$

$$\text{(SKIP)}\ \varGamma \vdash \mathbf{skip} \qquad \text{(ASSIGN)}\ \frac{\varGamma \vdash_{\textsc{f}} \textsc{e} : T' \qquad T' \preceq \varGamma_{\textsc{v}}(\textsc{v})}{\varGamma \vdash \textsc{v} := \textsc{e}\,\langle [\bullet \mapsto_d \varGamma_d(\bullet) \cup [\![\textsc{v}; \textsc{e}]\!]] \rangle}$$

$$\text{(VAR)}\ \frac{v \notin \varGamma_{\textsc{v}} \quad T \preceq \mathsf{Data}}{\varGamma \vdash v : T\,\langle [v \mapsto_{\textsc{v}} T] \rangle} \qquad \text{(NON-BL)}\ \frac{\varGamma \vdash p(\textsc{e}; \textsc{v})\,\langle \varDelta \rangle}{\varGamma \vdash \mathbf{await}\ p(\textsc{e}; \textsc{v})\,\langle \varDelta \rangle}$$

$$\text{(NEW)}\ \frac{\varGamma \vdash_{\textsc{f}} \textsc{e} : T \quad T \preceq \mathit{type}(\varGamma_{\mathcal{C}}(C).\mathit{Par}) \qquad \exists I \in \varGamma_{\mathcal{C}}(C).\mathit{Imp} \ \cdot \ I \preceq \varGamma_{\textsc{v}}(v)}{\varGamma \vdash v := \mathbf{new}\ C(\textsc{e})\,\langle [\bullet \mapsto_d \varGamma_d(\bullet) \cup [\![v; \textsc{e}]\!] \cup \{\langle C, \varGamma_{\mathcal{C}}(C).\mathit{Upg} \rangle\}] \rangle}$$

$$\text{(EXT)}\ \frac{\varGamma \vdash_{\textsc{f}} e : I \qquad \varGamma \vdash_{\textsc{f}} \textsc{e} : T \qquad \mathrm{matchext}(m, T, \textsc{v}, I, \varGamma_{\textsc{v}}(\mathit{self}), \varGamma)}{\varGamma \vdash e.m(\textsc{e}; \textsc{v})\,\langle [\bullet \mapsto_d \varGamma_d(\bullet) \cup [\![\textsc{e}; \textsc{v}]\!]] \rangle}$$

$$\text{(INT)}\ \frac{\varGamma \vdash_{\textsc{f}} \textsc{e} : T \qquad C' \in \mathrm{matchint}(\mathit{mtd}\,(m, \mathit{Internal}, T, \varGamma_{\textsc{v}}(\textsc{v}), \varepsilon), \varGamma_{\textsc{v}}(\mathit{self}), \varGamma)}{\varGamma \vdash m(\textsc{e}; \textsc{v})\,\langle [\bullet \mapsto_d \varGamma_d(\bullet) \cup [\![\textsc{e}; \textsc{v}]\!] \cup \{\langle C', \varGamma_{\mathcal{C}}(C').\mathit{Upg} \rangle\}] \rangle}$$

Figure 3. The type system, where • acts as a placeholder for values of type $\langle \tau_{\mathcal{C}} \times \mathsf{Nat} \rangle$, $[\![\textsc{e}]\!]$ returns a set of class names and upgrade numbers for the classes in which the attributes in an expression list $\textsc{e}$ are declared (relative to *self* in $\varGamma$), and *type* extracts the types of a declaration list.

5 Dynamic Class Upgrades

New interfaces, new classes, and class upgrades may update the running system. New interfaces and classes extend the system while class upgrades allow method redefinition as well as extending the class with new attributes, methods, interfaces, and superclasses. Modifications should not compromise the type safety of the running program; e.g., a method redefinition must preserve the signature so the class consistently supports its interfaces. In an open distributed setting, upgrades of classes and objects are not sequentialized; rather, upgrades propagate *asynchronously* through the network causing objects of different versions to coexist. Consequently, the order in which upgrades happen at runtime may differ from the order in which they were type checked. For upgrades with no syntactic dependencies, this overtaking does not affect runtime type safety. If there are syntactic dependencies between upgrades, the order of upgrades must respect these dependencies. The following kinds of system updates are considered:

Definition 5. *Systems are updated through the following operations:*

- *An* interface addition *is represented by a term* new-interface(N, R)*, where* N *is an interface name and* R *is an interface term.*

$$\text{(NEW-INTERFACE)}\ \frac{N \notin \mathit{\Gamma}_{\mathcal{I}} \qquad\qquad \mathit{\Gamma} + [N \mapsto_I R] \vdash R}{\mathit{\Gamma} \vdash \mathit{new\text{-}interface}\,(N, R)\,\langle N \mapsto_I R\rangle}$$

$$\text{(NEW-CLASS)}\ \frac{N \notin \mathit{\Gamma}_{\mathcal{C}} \qquad\qquad \mathit{\Gamma} + [\mathit{self} \mapsto_{\mathrm{v}} N] + [N \mapsto_C R]\ \vdash R\,\langle \mathit{\Delta}\rangle}{\mathit{\Gamma} \vdash \mathit{new\text{-}class}\,(N, R)\,\langle [N \mapsto_C R] + [\langle N, 1\rangle \mapsto_d (\mathit{\Delta}_d(\bullet) \setminus \{\langle N, 0\rangle\})]\rangle}$$

$$\text{(UP)}\ \frac{\begin{array}{l} \mathit{\Gamma} \vdash \mathit{self} : N; \mathit{\Gamma}_{\mathcal{C}}(N)\,\langle \mathit{\Gamma}'\rangle \qquad\qquad\qquad\qquad \forall I \in \mathit{Imp} \cdot I \in \mathit{\Gamma}_{\mathcal{I}} \\ \mathit{\Gamma} + \mathit{\Gamma}' \vdash \mathit{InhAttr}(\mathit{Inh}, \mathit{\Gamma}_{\mathcal{C}}); \mathit{Var}\ \langle \mathit{\Delta}\rangle \qquad\qquad \mathit{matchpar}(\mathit{\Gamma} + \mathit{\Gamma}', \mathit{Inh}) \\ \forall m \in \mathit{Mtd} \cdot\ \mathbf{if}\ m.\mathit{Nm} \in \mathit{\Gamma}_{\mathcal{C}}(N).\mathit{Mtd} \\ \quad \mathbf{then}\ \mathit{\Gamma} + \mathit{\Gamma}' + [N \mapsto_C \mathit{upg}(\mathit{\Gamma}_{\mathcal{C}}(N), 0, \epsilon, \mathit{Inh}, \epsilon, \mathit{Mtd} \setminus m)] + \mathit{\Delta} \vdash_r m\,\langle \mathit{\Delta}^m\rangle \\ \quad \mathbf{else}\ \ \mathit{\Gamma} + \mathit{\Gamma}' + [N \mapsto_C \mathit{upg}(\mathit{\Gamma}_{\mathcal{C}}(N), 0, \epsilon, \mathit{Inh}, \epsilon, \mathit{Mtd})] + \mathit{\Delta} \vdash m\,\langle \mathit{\Delta}^m\rangle\ \mathbf{fi} \\ \forall I \in \mathit{Imp} \cdot \forall m' \in \mathit{\Gamma}_{\mathcal{I}}(I).\mathit{Mtd}\ \cdot (\mathit{matchint}(m', (N; \mathit{Inh}), \mathit{\Gamma}) \neq \epsilon \\ \qquad\qquad \vee(\exists m \in \mathit{Mtd}(m'.\mathit{Nm}) \cdot \mathit{Sig}(m) \preceq \mathit{Sig}(m'))) \end{array}}{\begin{array}{l} \mathit{\Gamma} \vdash \mathit{upd}\,(N, \mathit{Imp}, \mathit{Inh}, \mathit{Var}, \mathit{Mtd})\ \langle\, [N \mapsto_C \mathit{upg}(\mathit{\Gamma}_{\mathcal{C}}(N), 1, \mathit{Imp}, \mathit{Inh}, \mathit{Var}, \mathit{Mtd})] \\ \qquad\qquad +[\langle N, \mathit{\Gamma}_{\mathcal{C}}(N).\mathit{Upg} + 1\rangle \mapsto_d \bigcup\limits_{m \in \mathit{Mtd}} \mathit{\Delta}_d^m(\bullet) \cup \{\langle N, \mathit{\Gamma}_{\mathcal{C}}(N).\mathit{Upg}\rangle\}]\rangle \end{array}}$$

$$\text{(MTD-RDEF)}\ \frac{\begin{array}{c} \mathit{Sig}(\mathit{mdef}) \preceq \mathit{Sig}(\mathit{\Gamma}_{\mathcal{C}}(\mathit{\Gamma}_{\mathrm{v}}(\mathit{self})).\mathit{Mtd}(\mathit{mdef}.\mathit{Nm})) \\ \mathit{\Gamma} + [\mathit{\Gamma}_{\mathcal{C}}(\mathit{\Gamma}_{\mathrm{v}}(\mathit{self}) \mapsto_C \mathit{upg}(\mathit{\Gamma}_{\mathcal{C}}(\mathit{\Gamma}_{\mathrm{v}}(\mathit{self}), 0, \epsilon, \epsilon, \epsilon, \mathit{mdef})] \vdash \mathit{mdef}\langle \mathit{\Delta}\rangle \end{array}}{\mathit{\Gamma} \vdash_r \mathit{mdef}\langle \mathit{\Delta}_d(\bullet)\rangle}$$

Figure 4. The type system for class upgrades. Here, $\vdash_r$ is used for type checking of redefined methods, and $Mtd(N)$ denotes the subset of methods in Mtd with name N.

- *A* class addition *is represented by a term* new-class(N, R)*, where N is a class name and R is a class term.*
- *A* class upgrade *is represented by a term* upd $(N, \mathrm{Imp}, \mathrm{Inh}, \mathrm{Var}, \mathrm{Mtd})$*, where N is the name of the class to be upgraded,* Imp *a list of interfaces,* Inh *a list of classes, defining additional superclasses to be inherited,* Var *a list of typed program variables, and* Mtd *a set of methods.*

Type checking class upgrades results in dependency conditions which ensure that system modifications do not violate the type safety of the running system. Given an upgrade of a class C in a well-typed program P, an upgrade is type checked based on the current typing environment $\mathit{\Gamma}$ of P: the mappings in $\mathit{\Gamma}$ are modified by upgrades. Thus, the upgraded versions of classes as accumulated in the environment resulting from a (successful) type checking, serve as the starting point of future updates.

5.1 Type Checking System Updates

The rules to type check new interfaces and classes, class upgrades, and method redefinitions are given in Figure 4. After type checking new interfaces and classes, the typing environment is extended. Let $\mathit{\Gamma}$ be the typing environment after type checking a well-typed program P. An upgrade of a class $C \in P$ is then type checked in $\mathit{\Gamma}$; i.e., $\mathit{\Gamma} \vdash \mathit{upd}\,(C, \mathit{Imp}, \mathit{Inh}, \mathit{Var}, \mathit{Mtd})\,\langle \mathit{\Gamma}_d' + {\mathit{\Gamma}_{\mathcal{C}}}'\rangle$, where ${\mathit{\Gamma}_{\mathcal{C}}}'$ is updates of the class representation in $\mathit{\Gamma}_{\mathcal{C}}$, computed by the auxiliary function *upg*, and $\mathit{\Gamma}_d'$ is dependency requirements to classes in P for the upgrade of C accumulated while type checking. The next update is type checked in $\mathit{\Gamma}+\mathit{\Gamma}_d'+{\mathit{\Gamma}_{\mathcal{C}}}'$.

Definition 6. *Let n be a natural number,* $\textsc{i}$ *a list of interfaces,* $\textsc{i}'$ *a list of classes,* $\textsc{v}$ *a list of variables, and* $\textsc{m}$ *a set of methods. The upgraded version of a class resulting from a class update is defined by the upg function:*

$$\begin{aligned} &\mathit{upg}(\mathit{class}\,(\mathit{Par}, \mathit{Upg}, \mathit{Imp}, \mathit{Inh}, \mathit{Var}, \mathit{Mtd}), n, \textsc{i}, \textsc{i}', \textsc{v}, \textsc{m}) \\ &\ = \mathit{class}\,(\mathit{Par}, \mathit{Upg} + n, \mathit{Imp}; \textsc{i}, \mathit{Inh}; \textsc{i}', \mathit{Var}; \textsc{v}, \mathit{Mtd} \oplus \textsc{m}) \end{aligned}$$

For class upgrades, the typing environment is reloaded for the upgrading class before type checking the upgrade elements with the rule (Up). By adding new interfaces, the class may provide new external services. For each new interface, the type system requires that the class provides, either by inheritance, by local declarations, or by the current upgrade, at least one type-correct method body for each method in the interface. The function *Sig* takes a method as argument and returns its signature, including the cointerface as an explicit in-parameter. If new superclasses are added, the inheritance list in $\Gamma_\mathcal{C}$ must be extended accordingly before type checking method bodies, as there might be internal calls to methods in the new superclasses. This also applies to methods, due to calls to methods introduced in the same upgrade. The function *matchpar* verifies that the formal and actual parameters of new inherited classes match, and that these classes are contained in the class mapping $\Gamma_\mathcal{C}$. Inherited attributes, as well as new object variables, will further extend the typing environment. For each method, the effect system of rule (Method) computes the dependencies associated with the method body. Finally, after the type analysis of the upgrade term of a class C, the $\Gamma_\mathcal{C}$ mapping is upgraded and the dependency mapping for the $(\Gamma_\mathcal{C}(C).\mathit{Upg} + 1)$'th upgrade of class C is constructed, which is a mapping from $\langle C, \Gamma_\mathcal{C}(C).\mathit{Upg} + 1\rangle$ to the dependencies identified by the type analysis of the upgrade term. For method redefinition, the rule (Mtd-rdef) ensures that the redefined method still satisfies the interface requirements implemented by the class. For purely internal methods, the new cointerface must be *Internal*.

At runtime, upgrades are asynchronous and may bypass each other. Hence, well-typed upgrades may give runtime errors if not applied in a type-correct order. We show that Γ_d, provided by the type system, helps to ensure that each upgrade is applied at an appropriate time: If both a class C' and a superclass C are updated, then upgrades will be applied at runtime in the order decided by the static type system, e.g., C is upgraded first if the upgrade of C' depends on the upgrade of C. However, upgrades that do not depend on each other may be applied in parallel. It is therefore necessary that $\Gamma_d(\langle C, u\rangle)$ is included as an argument to the runtime class upgrade $\langle C, u\rangle$. This is achieved by translating the update term $\mathit{upd}\,(C, \mathit{Imp}, \mathit{Inh}, \mathit{Var}, \mathit{Mtd})$ into the runtime message $\mathit{upgrade}\,(C, \mathit{Inh}, \mathit{Var}, \mathit{Mtd}, \Gamma_d(\langle C, \Gamma_C(C).\mathit{Upg}\rangle))$ where Γ is the environment obtained from type checking the update term. Note that the implements-clause is not needed after type checking.

5.2 Operational Semantics

The operational semantics of Creol is defined in rewriting logic (RL) [18] and is executable on the RL system Maude [6]. A rewrite theory is a 4-tuple (Σ, 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). 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 [18]. If rewrite rules apply to non-overlapping sub-configurations, the transitions may be performed in parallel. Consequently, concurrency is implicit in RL. Conditional rewrite rules $t \longrightarrow t'$ **if** *cond* are allowed, where the condition *cond* can be formulated as a conjunction of rewrites and equations that must hold for the main rule to apply.

A *system configuration* is a multiset combining Creol classes, objects, and messages. A Creol method call is reflected by a pair of messages, and object activity is organized around a *message queue* which contains incoming messages and a *process queue* which contains pending processes, i.e., remaining parts of method activations. The associative list constructor is written as ';', and the associative and commutative constructors for multisets and sets by whitespace. Representing argument positions by "_", terms $\langle \_ : Ob \,|\, Cl : \_, Pr : \_, PrQ : \_, Lvar : \_, Att : \_, Qu : \_ \rangle$ denote Creol objects, where *Ob* is the object identifier, *Cl* the *class identifier* which consists of a class name and *version number*, *Pr* the active process code, *PrQ* and *Qu* are multisets of pending processes and incoming messages with unspecified queue orderings, respectively, and *Lvar* and *Att* the local and object state, respectively. Terms $\langle \_ : Cl \,|\, Upd : \_, Inh : \_, Att : \_, Mtds : \_ \rangle$ represent Creol classes, where *Cl* is the class identifier, *Upd* the upgrade number, *Inh* a list of class identifiers, *Att* a list of attributes, and *Mtds* a set of methods. The class identifier for version n of class C is denoted $C\#n$. The rules for the static language constructs may be found in [13]. Focus here is on method binding and dynamic class constructs, given in Figure 5.

An *implicit inheritance graph* is used to facilitate dynamic reconfiguration mechanisms. The binding mechanism dynamically inspects the class hierarchy in the system configuration. When an invocation message $invoc(m, Sig, In)$ representing a call to a method m is found in the message queue of an object o of class $C\#n$, where *Sig* is the method signature as provided by the caller and *In* is the list of actual in-parameters, a message $bind(o, m, Sig, In)$ **to** $C\#n$ is generated. If m is defined locally in $C\#n$ with a matching signature, a process with the declared method code and local state is returned in a *bound* message. Otherwise, the *bind* message is retransmitted to the superclasses of C. Thus the *bind* message is sent from a class to its superclasses, dynamically unfolding the inheritance graph as far as needed and resulting in a *bound* message

$$\begin{aligned}
&\langle o\!:\!Ob \mid Cl : C\#n\rangle\ \langle o\!:\!Qu \mid Ev : \textsc{q}\ \mathit{invoc}(m, \mathit{Sig}, \mathit{In})\rangle\\
&\quad\longrightarrow \langle o\!:\!Ob \mid Cl : C\#n\rangle\ \langle o\!:\!Qu \mid Ev : \textsc{q}\rangle\ (\mathit{bind}(o, m, \mathit{Sig}, \mathit{In})\ \textbf{to}\ C\#n)
\end{aligned}$$

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

$$(\mathit{bound}(w)\ \textbf{to}\ o)\ \langle o\!:\!Ob \mid \mathit{PrQ} : \textsc{w}\rangle \longrightarrow \langle o\!:\!Ob \mid \mathit{PrQ} : w\ \textsc{w}\rangle$$

$$\begin{aligned}
&\mathit{new\text{-}class}(C, \textsc{i}, \textsc{a}, \textsc{m}, ((C'\#n)\ \textsc{r}))\ \langle C'\#n'\!:\!Cl \mid \mathit{Upd}\!:\!u\rangle\\
&\quad\longrightarrow \mathit{new\text{-}class}(C, \textsc{i}, \textsc{a}, \textsc{m}, \textsc{r})\ \langle C'\#n'\!:\!Cl \mid \mathit{Upd}\!:\!u\rangle\ \textbf{if}\ u \geq n
\end{aligned}$$

$$\mathit{new\text{-}class}(C, \textsc{i}, \textsc{a}, \textsc{m}, \varepsilon) \longrightarrow \langle C\#1\!:\!Cl \mid \mathit{Upd}\!:\!1, \mathit{Inh} : \textsc{i}, \mathit{Att} : \textsc{a}, \mathit{Mtds} : \textsc{m}, \mathit{Tok} : 1\rangle$$

$$\begin{aligned}
&\mathit{upgrade}\,(C, \textsc{i}, \textsc{a}, \textsc{m}, ((C'\#n)\ \textsc{r}))\ \langle C'\#n'\!:\!Cl \mid \mathit{Upd}\!:\!u\rangle\\
&\quad\longrightarrow \mathit{upgrade}\,(C, \textsc{i}, \textsc{a}, \textsc{m}, \textsc{r})\ \langle C'\#n'\!:\!Cl \mid \mathit{Upd}\!:\!u\rangle\ \textbf{if}\ u \geq n
\end{aligned}$$

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

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

$$\begin{aligned}
&\langle o\!:\!Ob \mid Cl : C\#n, Pr : \varepsilon\rangle\langle C\#n'\!:\!Cl \mid \mathit{Att} : \textsc{a}\rangle\\
&\quad= \langle o\!:\!Ob \mid Cl : C\#n', Pr : \varepsilon\rangle\ \langle C\#n'\!:\!Cl \mid \mathit{Att} : \textsc{a}\rangle\ (\mathit{getAttr}(o, \textsc{a})\ \textbf{to}\ C)\ \textbf{if}\ n' > n
\end{aligned}$$

$$(\mathit{gotAttr}(\textsc{a}')\ \textbf{to}\ o)\ \langle o\!:\!Ob \mid \mathit{Att} : \textsc{a}\rangle = \langle o\!:\!Ob \mid \mathit{Att} : \textsc{a}'\rangle$$

Figure 5. A RL specification of method binding and dynamic class upgrades.

returned to the object which generated the *bind* message. The auxiliary predicate $\mathit{match}(m, \mathit{Sig}, \textsc{m})$ is true if m is declared in $\textsc{m}$ with a signature Sig' such that $\mathit{Sig}' \preceq \mathit{Sig}$, and the function *get* fetches method m in the method set $\textsc{m}$ of the class and returns a process, resulting from the method activation. Values of the actual in-parameters *In* are stored in the local process state. The process is loaded into the internal process queue of the callee.

Class upgrades may be direct, or indirect through the upgrade of one of the superclasses. In order to control the upgrade propagation, class representations include an *upgrade number* and a *version number*; i.e., counters which record the number of times a class has been directly upgraded and (directly or indirectly) modified, respectively. When a class is upgraded, both its upgrade and version numbers are incremented. When a super-class of a class C is modified, the version number of C is incremented but the upgrade number of C does not change.

A *direct class upgrade* of a class C is realized through the insertion of a message $\mathit{upgrade}\,(C, \textsc{i}, \textsc{a}, \textsc{m}, \Gamma_d(\langle C, \Gamma_{\mathcal{C}}(C).\mathit{Upg}\rangle))$ in the system configuration at runtime, where $\textsc{i}$ is an inheritance list, $\textsc{a}$ a state, $\textsc{m}$ a set of method definitions, and $\Gamma_d(\langle C, \Gamma_{\mathcal{C}}(C).\mathit{Upg}\rangle)$ the set of upgrade requirements to classes in the runtime system directly derived from Γ, found by type checking. The upgrade of a class may not be applied unless these requirements are fulfilled. As upgrade is asynchronous, several upgrades may be pending in the runtime system, and the current upgrade may need to wait. A message $\mathit{upgrade}(C, \textsc{i}', \textsc{a}', \textsc{m}', \varepsilon)$, with an empty requirement set, does not have unverified dependencies, and the upgrade may be applied to C. The rule for *direct class upgrade* uses an operator $\oplus$ to overwrite the method set $\textsc{m}$ with the new or redefined methods in $\textsc{m}'$. During the upgrade, the upgrade and version numbers of the class are also incremented.

Indirect class upgrade propagates upgrade information to subclasses by means of an equation, so instances of the subclasses will acquire new state attributes. Note that by using an equation the indirect class upgrade happens in zero rewrite steps, which corresponds to temporarily locking the upgraded class.

The *upgrade of object instances* must ensure that new attributes are acquired before new code which may rely on new class attributes is evaluated. New object instances automatically get the new class attributes. However, the upgrade of existing object instances of the class must be closely controlled. Each time an object needs to evaluate a method, it requests the code associated with this method name. Problems may arise when executing new or redefined methods which rely on new attributes that are not presently available in the object. With recursive or nonterminating processes, 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, 7]. Consequently, it is too restrictive to wait for the completion of all pending methods before applying an upgrade. However, objects may reach *quiescent* states when the processor has been released and before any pending process has been activated. Any object which does not deadlock will 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. In the case of process termination or an inner suspension point, *Pr* is empty. The rule for *object upgrade* applies to quiescent states. Exploiting the implicit inheritance graph, attribute upgrade is handled by a message *getAttr*, similar to *bind*, which recursively extends the object state $\textsc{a}$ and results in a message $\mathit{gotAttr}(\textsc{a}')$. The new object state $\textsc{a}'$ finally replaces $\textsc{a}$. The use of equations corresponds to locking the object.

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.

5.3 Type-Safe Execution with Dynamic Class Upgrades

The problem of type-safe execution of programs is now addressed. We prove that errors such as method-not-understood do not occur at runtime, even with the proposed dynamic class construct. A type soundness theorem for Creol without dynamic classes was shown in [15]: *Runtime type errors do not occur for well-typed programs.* The theorem implies that runtime assignments to program variables, object creation, and method invocations are type-correct. The proof is by induction over the length of the execution sequence as given by the operational semantics. However, dynamic upgrades as considered in this paper introduce runtime changes as the state adapts to the upgrades. By reasoning about the type system and operational semantics, the following properties are proved for the class upgrade mechanism of this paper:

Lemma 1. *A well-typed class upgrade does not affect the execution of code of existing processes in an object.*

Lemma 2. *The execution of a method activation from a new version of an object's class will not begin before the object's state is updated.*

Lemma 3. *Let Γ be the typing environment for a well-typed program after a series of upgrades, including the upgrade $\langle C, u\rangle$. The upgrade $\langle C, u\rangle$ is applicable iff the runtime structure satisfies $\Gamma_d(\langle C, u\rangle)$.*

Lemma 4. *The execution of processes introduced in a well-typed upgrade will not cause runtime type errors.*

Lemma 4 follows from Lemmas 2 and 3. Lemmas 1 and 4 show that variable assignments, object creation, and method invocations are type-correct when classes are upgraded, for old and new processes, respectively. A type soundness property for Creol with class upgrades can now be proved by induction over the length of execution sequences, extending the proof for the language without dynamic classes. Lemmas 1 and 4 are used for the application of class upgrades:

Theorem 1 (Type soundness). *Well-typed upgrades do not introduce runtime type errors in well-typed programs.*

6 Related Work

Availability during reconfiguration 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 [22], functional [3], and object-oriented [8] languages. These approaches consider the upgrade of single type declarations, procedures, objects, or components in the sequential setting. Reclassification in Fickle [8] is based on a type system which guarantees type safety when an object changes its class. Fickle has been extended to multithreading [7], but restrictions to runtime reclassification are needed; e.g., an object with a nonterminating (recursive) method will not be reclassified.

Version control systems aim at a more modular upgrade support. Some approaches allow multiple module versions to coexist after an upgrade [2, 3, 9–11], while others only keep the last version by doing a global update or "hot-swapping" [1, 5, 17, 19]. The approaches also differ in their treatment of active behavior, which may be disallowed [5,10,17,19], delayed [1,7], or supported [11,22]. Approaches based on global update mostly disallow upgrades of active modules. An upgrade system for type declarations and procedures in active code is proposed in [22] for (sequential) C. Type-safe updates occur at annotated program points found by the type system. However, the approach is synchronous as upgrades which cannot be applied immediately will fail.

Dynamic class constructs support modular upgrades. The approach of Hjálmtýsson and Gray [11] for C++, based on proxy classes which link to the actual classes (reference indirection), supports multiple versions of each class. Existing instances are not upgraded, so the activity in existing objects is uninterrupted. Existing approaches for Java, either using proxies [19] or modifying the Java virtual machine [17], are based on global upgrade and are not applicable to active objects. In [19], each class version supports the same interfaces. New interfaces can only be introduced by adding new classes. In [5] the ordering of upgrades are serialized and in [17] invalid upgrades are handled by exceptions.

Automatic upgrade based on lazy global update is addressed in [1] for distributed objects and in [5] for persistent object stores. Here the object instances of upgraded classes are upgraded, but inheritance and (nonterminating) active code are not addressed, which limits the effect of class upgrade. 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. Moreover, the type system handles upgrade dependencies among distributed objects.

7 Conclusion

In this paper a construct for dynamic class upgrades in Creol is presented, including its type system and operational semantics, which allows method redefinition as well as extending classes with new attributes, methods, superclasses, and interfaces, in the running system. By adding new interfaces, classes may provide new external services, while the redefinition of methods may improve existing ones. Our approach exempts programmers from handling the different version numbers of classes when writing upgrade codes.

To address open distributed systems with concurrent objects, we consider an asynchronous update mechanism where upgrade overtaking is possible in the runtime system, and allow objects of different versions to coexist. A successful introduction of upgrades in this setting requires both type checking and careful timing of when the upgrades are applied. Runtime errors would occur if upgrades are applied at a bad time. The type system captures upgrade dependencies and enforces an ordering of upgrades. If the type checking of an upgrade succeeds, an *effect* system provides a list of dependencies for the upgrade. This list of dependencies is used by the runtime system to ensure that dependent upgrades are applied in an order which preserves type correctness, while independent upgrades may be performed simultaneously. Furthermore, it is shown that well-typed runtime upgrades do not introduce type errors. In future work we plan to extend the dynamic construct proposed in this paper with type-safe mechanisms for removing attributes and method definitions, using similar techniques.

References

1. S. Ajmani, B. Liskov, and L. Shrira. Scheduling and simulation: How to upgrade distributed systems. In *Hot Topics in Op. Sys. (HotOS-IX)*, pages 43–48, 2003.
2. J. L. Armstrong and S. R. Virding. Erlang - an experimental telephony programming language. In *XIII International Switching Symposium*, June 1990.
3. G. Bierman, M. Hicks, P. Sewell, and G. Stoyle. Formalizing dynamic software updating. In *Unanticipated Software Evolution (USE)*, May 2003.
4. T. Bloom. *Dynamic Module Replacement in a Distributed Programming System*. PhD thesis, MIT, 1983. Also available as MIT LCS Tech. Report 303.
5. C. Boyapati *et al.* Lazy modular upgrades in persistent object stores. In *OOPSLA 2003*, pages 403–417. ACM Press, 2003.
6. M. Clavel *et al.* Maude: Specification and programming in rewriting logic. *Theoretical Computer Science*, 285:187–243, Aug. 2002.
7. F. Damiani, M. Dezani-Ciancaglini, and P. Giannini. Re-classification and multithreading: Fickle$_{\mathrm{MT}}$. In *Symp. Applied Computing (SAC'04)*. ACM Press, 2004.
8. S. Drossopoulou, F. Damiani, M. Dezani-Ciancaglini, and P. Giannini. More dynamic object re-classification: Fickle$_{\mathrm{II}}$. *ACM TOPLAS*, 24(2):153–191, 2002.
9. D. Duggan. Type-Based hot swapping of running modules. In *Intl. Conf. Functional Programming (ICFP-01)*, *ACM SIGPLAN* 36(10), pages 62–73, Sept. 2001.
10. 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.
11. G. Hjálmtýsson and R. S. Gray. Dynamic C++ classes: A lightweight mechanism to update code in a running program. In *Proc. USENIX Tech. Conf.*, May 1998.
12. C. R. Hofmeister and J. M. Purtilo. A framework for dynamic reconfiguration of distributed programs. Tech. Report CS-TR-3119, Univ. of Maryland, 1993.
13. E. B. Johnsen and O. Owe. A dynamic binding strategy for multiple inheritance and asynchronously communicating objects. *Proc. FMCO'04*, LNCS 3657. Springer, 2005.
14. E. B. Johnsen, O. Owe, and I. Simplot-Ryl. A dynamic class construct for asynchronous concurrent objects. In *Proc. FMOODS*, LNCS 3535. Springer, June 2005.
15. E. B. Johnsen, O. Owe, and I. C. Yu. Creol: A type-safe object-oriented model for distributed concurrent systems. Res. Rep. 327, Ifi, Univ. of Oslo, 2005.
16. J. Kramer and J. Magee. The Evolving Philosophers Problem: Dynamic change management. *IEEE Trans. on Software Engineering*, 16(11):1293–1306, Nov. 1990.
17. S. Malabarba, R. Pandey, J. Gragg, E. Barr, and J. F. Barnes. Runtime support for type-safe dynamic Java classes. In *Proc. ECOOP*, LNCS 1850. Springer, 2000.
18. J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96:73–155, 1992.
19. A. Orso, A. Rao, and M. J. Harrold. A technique for dynamic updating of Java software. In *Software Maintenance (ICSM 2002)*, pages 649–658. IEEE Press, 2002.
20. B. C. Pierce. *Types and Programming Languages*. The MIT Press, 2002.
21. C. A. N. Soules *et al.* System support for online reconfiguration. In *Proc. USENIX Tech. Conf.*, pages 141–154, 2003.
22. G. Stoyle, M. Hicks, G. Bierman, P. Sewell, and I. Neamtiu. *Mutatis Mutandis*: Safe and flexible dynamic software updating. In *Proc. POPL*, ACM Press, 2005.