Dynamic Classes: Modular Asynchronous Evolution of Distributed Concurrent Objects *

Einar Broch Johnsen[1], Marcel Kyas[2], and Ingrid Chieh Yu[1]

[1] Department of Informatics, University of Oslo, Norway
{einarj,ingridcy}@ifi.uio.no
[2] Department of Computer Science, Freie Universität Berlin, Germany
marcel.kyas@fu-berlin-de

Abstract. Many long-lived and distributed systems must remain available yet evolve over time, due to, e.g., bugfixes, feature extensions, or changing user requirements. To facilitate such changes, formal methods can help in modeling and analyzing runtime software evolution. This paper presents an executable object-oriented modeling language which supports runtime software evolution. The language, based on Creol, targets distributed systems by active objects, asynchronous method calls, and futures. A dynamic class construct is proposed in this setting, providing an asynchronous and modular upgrade mechanism. At runtime, class redefinitions gradually upgrade existing instances of a class and of its subclasses. An upgrade may depend on previous upgrades of other classes. For asynchronous runtime upgrades, the static picture may differ from the actual runtime system. An operational semantics and a type and effect system are given for the language. The type analysis of an upgrade infers and collects dependencies on previous upgrades. These dependencies are exploited as runtime constraints to ensure type safety.

1 Introduction

Many long-lived distributed systems require continuous system availability, but still need to change their code due to bugfixes as well as new, improved, or redundant functionality. Examples of such systems are found in, e.g., financial transactions, aeronautics and space missions, biomedical sensors, and telephony and Internet services. For these systems, code changes must happen at runtime. In large distributed systems, runtime updates need to be applied in an asynchronous and modular manner, and propagate gradually through the distributed system. A challenge for software upgrade systems is to balance flexibility, robustness, and user-friendliness. An appropriate upgrade system should propagate upgrades automatically, provide means to control *when* components are upgraded, and ensure the availability of system services during the upgrade [1,20]. In order

* This research is partly funded by the EU projects IST-33826 CREDO: Modeling and Analysis of Evolutionary Structures for Distributed Services (http://credo.cwi.nl) and FP7-231620 HATS: Highly Adaptable and Trustworthy Software using Formal Methods (http://www.hats-project.eu).

to ensure that upgrades are correct and result in foreseen changes, formal models and analysis methods for runtime software evolution are needed.

This paper presents a modeling language which supports the runtime evolution of distributed object-oriented systems. The language extends Creol [17], an executable formalism in which distributed concurrent objects communicate by asynchronous method calls and futures [7, 8, 22], with *dynamic class operations.* These may introduce new functionality and interfaces for classes, change data structures and implementations for existing functionality, and remove legacy code. Dynamic class operations provide a *modular* form of software evolution because upgrades to a class C apply to all existing instances of C and of its subclasses. Compared to previous approaches [6, 13, 18, 19], our approach supports the gradual upgrade of active objects and upgrade operations propagate asynchronously through the system. It is a challenge for formal methods to reason about the runtime evolution of distributed systems. This paper focuses on ensuring type safety at runtime as class definitions evolve.

An operational semantics and type system for dynamic class operations are introduced and integrated with the semantics and type system of Creol. Creol is type-safe in the sense that runtime type errors do not occur for well-typed programs; in particular, method binding always succeeds. We show that well-typed dynamic class operations maintain this property. As classes gradually evolve, a type-safe upgrade of one class may require that an upgrade of another class has already been applied; e.g., when new code contains calls to methods introduced in a previous upgrade. Upgrades may be arbitrarily delayed in the asynchronous setting, so upgrades injected into the system in one order may be applied in another. This causes a discrepancy between the static system view, as provided by a typing environment for dynamic class operations, and the situation at runtime. We develop a type and effects system [2] to analyze dynamic class upgrades and to automatically infer and collect dependencies between class upgrades. Thus, the dependencies of an upgrade operation need not be provided by the modeler. A characterizing feature of our approach is that the dependencies inferred during type analysis are imposed as constraints on the applicability of a particular upgrade at runtime. This may delay certain upgrade operations at runtime to ensure that execution remains type safe. This paper presents the proposed dynamic class operations for a kernel language, but the approach is supported in the complete Creol language. The type system and operational semantics of this paper have been implemented and integrated with Creol's execution platform.

Paper overview. Sect. 2 presents the kernel language, its type system, and semantics. Sect. 3 provides an example of dynamic class upgrades. Sect. 4 introduces operations for dynamic class upgrades, their type system, and semantics. Sect. 5 discusses related work, and Sect. 6 concludes the paper.

2 A Language for Distributed Concurrent Objects

Consider a kernel language for distributed concurrent objects, similar to, e.g., Featherweight Java [15]. The language targets distributed systems by supporting

asynchronous method calls and futures (i.e., returns from asynchronous calls). In contrast to Java each concurrent object encapsulates its state; i.e., all external manipulation of the object state is through calls to the object's methods. In addition, objects execute concurrently: each object has a processor which executes the processes of that object. Processes in different objects execute in parallel. A process corresponds to the activation of a method. Only one process may be active in an object at a time; the other processes in the object are *suspended*. We distinguish between *blocking* a process and *releasing* a process. Blocking is used for synchronization and stops the execution of the process, but does not let a suspended process resume. Releasing a process suspends the execution of that process and lets a suspended process resume. Thus, if a process is blocked there is no execution in the object, if it is released another process in the object may execute. Although processes need not terminate, the execution of several processes may be combined using *release points* within method bodies. At a release point, the active process may be released and a suspended process may resume.

Method calls are asynchronous and the result of a call is stored in a future, which may be read or polled. Return values are accessed by need; i.e., the execution blocks if attempting to read from a future without a return value. In contrast, polling a future never blocks. The scheduling of processes at release points is influenced by await-statements with Boolean guards, including the polling of futures. If a guard evaluates to false, the process is released. Only a process whose guard evaluates to true may resume execution. Remark that release points make it straightforward to combine active (i.e., nonterminating) and reactive processes in an object. Thus, an object may behave both as a client and as a server while abstracting from the exact interleaving of these roles.

The behavior of an object may depend on its context of interaction; we let object variables (references) be typed by *interfaces*. These contain method signatures and provide context-dependent encapsulation, as different sets of methods may be available through different interfaces. Variables typed by different interfaces may refer to the same object. A class *implements* an interface I if its instances may be typed by I. A class may implement several interfaces and different classes may provide different implementations of the same interface. Reasoning control is ensured by substitutability at the level of interfaces: *an object supporting an interface I may be replaced by another object supporting I or a subinterface of I* in a context depending on I. This substitutability is reflected in the semantics by the fact that late binding applies to all external method calls, as the runtime class of the object reference is not in general statically known.

The *syntax* is given in Fig. 1. We emphasize the differences with Java. A program P is a list of interface and class definitions, followed by a method body. To illustrate the generality of the dynamic class construct, a class may inherit from a list of superclasses (possibly just Object), extending these with additional fields $\overline{f}$ and methods $\overline{M}$. *Expressions* e are standard apart from the asynchronous method call $e!m(\overline{e})$ and the (blocking) read operation $v.\mathsf{get}$. *Statements* s are standard apart from release points $\mathsf{await}\ g$ and release. *Guards* g are conjunctions of Boolean expressions b and polling operations $v?$ on futures v. When the guard

$$
\begin{array}{llll}
P & ::= \overline{D}\ \overline{L}\ \{\overline{T\ x}; sr\} & D & ::= \mathsf{interface}\ I\ \mathsf{extends}\ \overline{I}\ \{\overline{M_s}\} \\
sr & ::= s; \mathsf{return}\ e & L & ::= \mathsf{class}\ C\ \mathsf{extends}\ \overline{C}\ \mathsf{implements}\ \overline{I}\ \{\overline{T\ f}; \overline{M}\} \\
v & ::= f \mid x & M & ::= M_s\{\overline{T\ x}; sr\} \\
b & ::= \mathsf{true} \mid \mathsf{false} \mid v & e & ::= v \mid \mathsf{new}\ C(\,) \mid e.\mathsf{get} \mid e!m(\overline{e}) \mid \mathsf{null} \\
T & ::= I \mid \mathsf{bool} \mid \mathsf{fut}(T) & s & ::= v := e \mid \mathsf{await}\ g \mid \mathsf{skip} \mid s; s \mid \mathsf{if}\ g\ \mathsf{then}\ s\ \mathsf{fi} \mid \mathsf{release} \\
M_s & ::= T\ m\ (\overline{T\ x}) & g & ::= b \mid v? \mid g \wedge g
\end{array}
$$

Fig. 1. The language syntax. Variables v are fields (f) or local variables (x), C is a class name, and I an interface name.

in an await statement evaluates to false, the statement gets preceded by a release, otherwise it becomes a skip. The release statement suspends the active process.

2.1 Typing

Type analysis is done by a type and effect system [2] in the context of a *mapping family*, defined as follows: 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 is defined by $\emptyset_i = \varepsilon$ and $(\Gamma + [n{\mapsto_{i'}}d])_i = \textbf{if } (i = i') \textbf{ then } \Gamma_i + [n{\mapsto_i}d] \textbf{ else } \Gamma_i$. The *application* for the indexed mapping Γ_i, is $\varepsilon(n) = \bot$, and $(\Gamma_i + [n{\mapsto_i}d])(n') = \textbf{if } (n = n') \textbf{ then } d \textbf{ else } \Gamma_i(n')$.

The typing context uses four indexes; the mappings $\Gamma_{\mathcal{I}}$ and $\Gamma_{\mathcal{C}}$ map interface and class names to interface and class declarations, and Γ_v maps program variable names to types. In the absence of class upgrades, $\Gamma_{\mathcal{I}}$ and $\Gamma_{\mathcal{C}}$ correspond to *static tables*. The subtype relation $T_1 \preceq T_2$ is defined by interface inheritance. Only declarations extend Γ_v. Some auxiliary functions are defined on a mapping family Γ. The field declarations in a class C and its superclasses are collected by $\mathit{attr}(C, \Gamma)$ and $\mathit{implements}(C, I, \Gamma)$ matches signatures for methods declared in an interface I to those in C (to check that C provides bodies for the declarations of I). We assume for simplicity that variable declarations $\overline{T\ x}$ are well-typed and denote by $[\overline{x}{\mapsto_v}\overline{T}]$ the associated mapping (built from the bindings $[x{\mapsto_v}T]$).

Finally, there is a mapping of *dependencies* $\Gamma_d : \mathsf{Dep} \to \mathsf{Set}[\mathsf{Dep}]$, where the type Dep consist of pairs of class names and natural numbers. An upgrade of a class C can be uniquely identified by a natural number; e.g., $\langle C, 5\rangle$ represents the fifth upgrade of C. Elements in $\Gamma_d(\langle C, u\rangle)$ will represent classes on which an upgrade u of a class C depends; these dependencies are inferred from the current class table by the type analysis, and exploited for dynamic classes in Sect. 4.

The type rules are given in Fig. 2. Judgments have the form $\Gamma \vdash e : T\,\langle\Sigma\rangle$ and $\Gamma \vdash s\,\langle\Sigma\rangle$, where Γ is the typing environment and $\Sigma : \mathsf{Set}[\mathsf{Dep}]$ the effect. To simplify the presentation, we assume that method declarations in interfaces are unique and well-typed and omit the analysis of interfaces. Furthermore, the (straightforward) definitions of auxiliary functions on Γ are omitted.

The rules (Poll) and (Get) for operations on futures convert types from T to $\mathsf{fut}(T)$. Let $\mathit{interfaces}(\Gamma_{\mathcal{C}}(C))$ denote the interfaces of C as declared in $\Gamma_{\mathcal{C}}$. Rule (New) shows the connection between the type of the variable and the interfaces

$$\frac{\Gamma \vdash v : \mathsf{fut}(T)\,\langle\Sigma\rangle}{\Gamma \vdash v? : \mathsf{bool}\,\langle\Sigma\rangle}\ (\textsc{Poll}) \qquad \frac{\Gamma \vdash v : \mathsf{fut}(T)\,\langle\Sigma\rangle}{\Gamma \vdash v.\mathsf{get} : T\,\langle\Sigma\rangle}\ (\textsc{Get})$$

$$\frac{\exists T' \in \mathit{interfaces}(\Gamma_{\mathcal{C}}(C)) \cdot T' \preceq T}{\Gamma \vdash \mathsf{new}\ C(\,) : T\,\langle(C, \mathit{curr}(C,\Gamma))\rangle}\ (\textsc{New}) \qquad \frac{}{\Gamma \vdash \mathsf{skip}}\ (\textsc{Skip})$$

$$\frac{\Gamma \vdash \overline{e} : T\,\langle\Sigma\rangle \quad \exists C \in \mathit{matchint}(m, T \to T', \Gamma_v(\mathsf{this}), \Gamma_{\mathcal{C}})}{\Gamma \vdash \mathsf{this}!m(\overline{e}) : \mathsf{fut}(T')\,\langle\Sigma \cup (C, \mathit{curr}(C,\Gamma))\rangle}\ (\textsc{IntCall})$$

$$\frac{\Gamma \vdash \overline{e} : T\,\langle\Sigma_1\rangle \quad \Gamma \vdash e : I\,\langle\Sigma_2\rangle \quad \mathit{matchext}(m, T \to T', I, \Gamma_{\mathcal{I}})}{\Gamma \vdash e!m(\overline{e}) : \mathsf{fut}(T')\,\langle\Sigma_1 \cup \Sigma_2\rangle}\ (\textsc{ExtCall}) \qquad \frac{I \in \mathit{dom}(\Gamma_{\mathcal{I}})}{\Gamma \vdash \mathsf{null} : I}\ (\textsc{Null})$$

$$\frac{\Gamma(v) = T}{\Gamma \vdash v : T\,\langle[\![v]\!]\rangle}\ (\textsc{Var}) \qquad \frac{\Gamma \vdash e : T'\,\langle\Sigma\rangle \qquad T' \preceq \Gamma_v(v)}{\Gamma \vdash v := e\,\langle[\![v]\!] \cup \Sigma\rangle}\ (\textsc{Assign})$$

$$\frac{\Gamma \vdash g_1 : \mathsf{bool}\,\langle\Sigma_1\rangle \quad \Gamma \vdash g_2 : \mathsf{bool}\,\langle\Sigma_2\rangle}{\Gamma \vdash g_1 \wedge g_2 : \mathsf{bool}\,\langle\Sigma_1 \cup \Sigma_2\rangle}\ (\textsc{And})$$

$$\frac{\Gamma \vdash g : \mathsf{bool}\,\langle\Sigma\rangle}{\Gamma \vdash \mathsf{await}\ g\,\langle\Sigma\rangle}\ (\textsc{Await}) \qquad \frac{}{\Gamma \vdash \mathsf{release}}\ (\textsc{Release})$$

$$\frac{\Gamma \vdash s\,\langle\Sigma_1\rangle \quad \Gamma \vdash s'\,\langle\Sigma_2\rangle}{\Gamma \vdash s; s'\,\langle\Sigma_1 \cup \Sigma_2\rangle}\ (\textsc{Composition}) \qquad \frac{\Gamma \vdash b : \mathsf{bool}\,\langle\Sigma_1\rangle \quad \Gamma \vdash s\,\langle\Sigma_2\rangle}{\Gamma \vdash \mathsf{if}\ b\ \mathsf{then}\ s\ \mathsf{fi}\,\langle\Sigma_1 \cup \Sigma_2\rangle}\ (\textsc{Conditional})$$

$$\frac{\Gamma' = \Gamma + [\overline{x} \mapsto_v \overline{T}] + [\overline{x'} \mapsto_v \overline{T'}] \quad \Gamma' \vdash e : T'\,\langle\Sigma_1\rangle \quad \Gamma' \vdash s\,\langle\Sigma_2\rangle}{\Gamma \vdash T'\ m\ (\overline{T\ x})\{\overline{T'\ x'}; s; \mathsf{return}\ e\}\,\langle\Sigma_1 \cup \Sigma_2\rangle}\ (\textsc{Method})$$

$$\frac{\Gamma + [\overline{x} \mapsto_v \overline{T}] \vdash s \quad \forall L \in \overline{L} \cdot \Gamma_{\mathcal{I}} + \Gamma_{\mathcal{C}} + \Gamma_d^L \vdash L}{\Gamma_{\mathcal{I}} + \Gamma_{\mathcal{C}} + \bigcup_{L \in \overline{L}} \Gamma_d^L \vdash \overline{L}\,\{\overline{T\ x}; s; \mathsf{return}\ \mathit{true}\}}\ (\textsc{Program})$$

$$\frac{\forall M \in \overline{M}\ \cdot\ \Gamma + [\mathsf{this} \mapsto_v C] + [\mathit{attr}(C,\Gamma)] \vdash M\,\langle\Sigma^M\rangle \quad \forall I \in \overline{I} \cdot \mathit{implements}(C, I, \Gamma)}{\Gamma + [\langle C, 0\rangle \mapsto_d \bigcup_{M \in \overline{M}} \Sigma^M] \vdash \mathsf{class}\ C\ \mathsf{extends}\ \overline{C}\ \mathsf{implements}\ \overline{I}\ \{\overline{T\ f}; \overline{M}\}}\ (\textsc{Class})$$

Fig. 2. The type and effect system. Judgments for the Boolean constants true and false are similar to (Release). We omit empty effects; e.g., $\Gamma \vdash e\,\langle\emptyset\rangle$ is written $\Gamma \vdash e$.

of the class; the typing of a class instance depends on the context. The function $\mathit{curr}(C, \Gamma)$ identifies the current version number of a class C. In (IntCall) the auxiliary predicate *matchint*, given a method name, signature, and class, checks that an internal invocation may be bound in the class mapping. Similarly, in (ExtCall), *matchext* checks that the interface of the callee can bind the external call. Any interface can type null in (Null). For a variable v, let $[\![v]\!] : \mathsf{Dep}$ denote the class in which v is declared and its version number (which is easily retrieved from the typing environment). The effect of the analysis of expressions and guards is a set of dependencies to versions of the current class and its superclasses.

In rule (Method), local declarations extend the typing environment used for statements in the method body. The dependencies from different statements are accumulated in (Composition). The effect of the analysis of a method is the set of all dependencies from the body of the method. In (Class), this is bound to the class name, the context is extended with fields, and each method is typechecked. For each method M in rule (Class), the dependencies of M are stored in the effect Σ^M. Thus, Γ_d maps the dependencies of the initial version of C to the dependencies accumulated from the type analysis of the class; i.e., $\langle C, 0\rangle \mapsto_d \bigcup_{M \in \overline{M}} \Sigma^M$. In (Program), a program is type checked in the context $\Gamma_{\mathcal{I}} + \Gamma_{\mathcal{C}}$. Here, Γ_d^L denotes the dependency mapping derived for class L in the program.

$$
\begin{array}{rcl@{\qquad}rcl}
\mathit{config} & ::= & \epsilon \mid \mathit{class} \mid \mathit{object} \mid \mathit{msg} \mid \mathit{config}\ \mathit{config} & o & ::= & (\mathit{oid}, C\#n) \\
\mathit{class} & ::= & (C\#n, \mathit{vs}, \mathit{impl}, \mathit{inh}, \mathit{ob}, \mathit{fds}, \mathit{mtds}) & \mathit{fds} & ::= & \overline{T\ v\ \mathit{val}} \\
\mathit{object} & ::= & (o, \mathit{pv}, \mathit{processQ}, \mathit{fds}, \mathit{active}) & \mathit{active} & ::= & \mathit{process} \mid \mathsf{idle} \\
\mathit{processQ} & ::= & \epsilon \mid \mathit{process} \mid \mathit{processQ}\ \mathit{processQ} & \mathit{mc} & ::= & \mathit{oid}.m(\overline{\mathit{val}}) \\
\mathit{msg} & ::= & (\mathit{fid}, \mathit{mc}, \mathit{mode}, \mathit{val}) \mid (\mathit{bind}, \overline{C}, \mathit{fid}, \mathit{mc}) & \mathit{val} & ::= & \mathit{oid} \mid \underline{\mathit{fid}} \mid \mathsf{null} \mid b \\
 & & \mid (\mathit{bound}, o, \mathit{process}) & \mathit{mtd} & ::= & T\ m(\overline{T\ x})\{\mathit{process}\} \\
\mathit{process} & ::= & (\mathit{fds}, \mathit{sr}) \mid \mathsf{error} & \mathit{mtds} & ::= & \epsilon \mid \mathit{mtd} \mid \mathit{mtds}\ \mathit{mtds}
\end{array}
$$

Fig. 3. Syntax for runtime configurations; *oid* and *fid* are object and future identifiers.

2.2 Context-Reduction Semantics

The semantics is given by a small-step reduction relation on *configurations* of objects, classes, and futures (see Fig. 3). To accommodate upgrades in Sect. 4, stage and version numbers are introduced in the semantics. A *class* has an id (i.e., a name and a *stage number* n: Nat, which changes when the class or one of its superclasses is upgraded), a *version number* vs: Nat (which changes only when the class itself is upgraded), a list of interfaces, a list of superclasses, a set of object ids, a set of fields with default values, and a set of methods. Default values for types are given by a function *default* (e.g., $\mathit{default}(I) = \mathsf{null}$, $\mathit{default}(\mathsf{bool}) = \mathsf{false}$, and $\mathit{default}(!T) = \mathsf{null}$). An *object* has an id *oid*, a class with a stage number $C\sharp n$, a process version set $pv : \mathsf{Set}[\mathit{fid} \times \mathsf{Nat}]$, a queue *pq* of suspended processes, fields *fds*, and an active process. In an object o, *pv* tracks the stage of the class for (pending) method activations on o: these may be either internal calls or incoming requests. The idle process indicates that no method is active in the object and error that method binding has failed. A *future* $(\mathit{fid}, \mathit{mc}, \mathit{mode}, \mathit{val})$ captures the state of a method call: initially *sleeping*, then *active*, and finally, it becomes *completed* and stores the result from the call. Let $\mathit{mode} \in \{\mathsf{s}, \mathsf{a}, \mathsf{c}\}$ represent these states. The *initial configuration* of a program $\overline{L}\ \{\overline{T\ x}; \mathit{sr}\}$ has classes and one object $(o, \emptyset, \epsilon, \epsilon, (\overline{T\ x\ \mathit{default}(T)}), \mathit{sr})$.

Reduction takes the form of a relation $\mathit{config} \rightarrow \mathit{config}'$. The main rules are given in Fig. 4. The context reduction semantics decomposes a statement into a reduction context and a redex, and reduces the redex [12]. *Reduction contexts* are method bodies M, statements S, expressions E, and guards G with a single hole denoted by $\bullet$:

$$
\begin{array}{lll}
M ::= \bullet \mid S; \mathsf{return}\ e \mid \mathsf{return}\ E & & S ::= \bullet \mid v := E \mid S; s \mid \mathsf{if}\ G\ \mathsf{then}\ s_1\ \mathsf{fi} \\
E ::= \bullet \mid E.\mathsf{get} \mid E!m(\overline{e}) \mid \mathit{oid}!m(\overline{\mathit{val}}, E, \overline{e}) & & G ::= \bullet \mid E? \mid G \wedge g \mid b \wedge G
\end{array}
$$

Redexes reduce in their respective contexts; i.e., body-redexes in M, stat-redexes in S, expr-redexes in E, and guard-redexes in G. Redexes are defined as follows:

$$
\begin{array}{ll}
\mathit{body\text{-}redexes} & ::= \mathsf{return}\ \mathit{val} \\
\mathit{stat\text{-}redexes} & ::= x := \mathit{val} \mid f := \mathit{val} \mid \mathsf{await}\ g \mid \mathsf{skip}; s \mid \mathsf{if}\ b\ \mathsf{then}\ s\ \mathsf{else}\ s\ \mathsf{fi} \mid \mathsf{release} \\
\mathit{expr\text{-}redexes} & ::= x \mid f \mid \mathit{fid}.\mathsf{get} \mid \mathit{oid}!m(\overline{\mathit{val}}) \mid \mathsf{new}\ C() \\
\mathit{guard\text{-}redexes} & ::= \mathit{fid}? \mid b \wedge g
\end{array}
$$

Filling the hole of a context M with a redex r is denoted $M[r]$. Before evaluating the expression e in the method body $s; \mathsf{return}\ e$, the body will be reduced to $\mathsf{skip}; \mathsf{return}\ e$. For simplicity, we elide the skip and write just $\mathsf{return}\ e$.

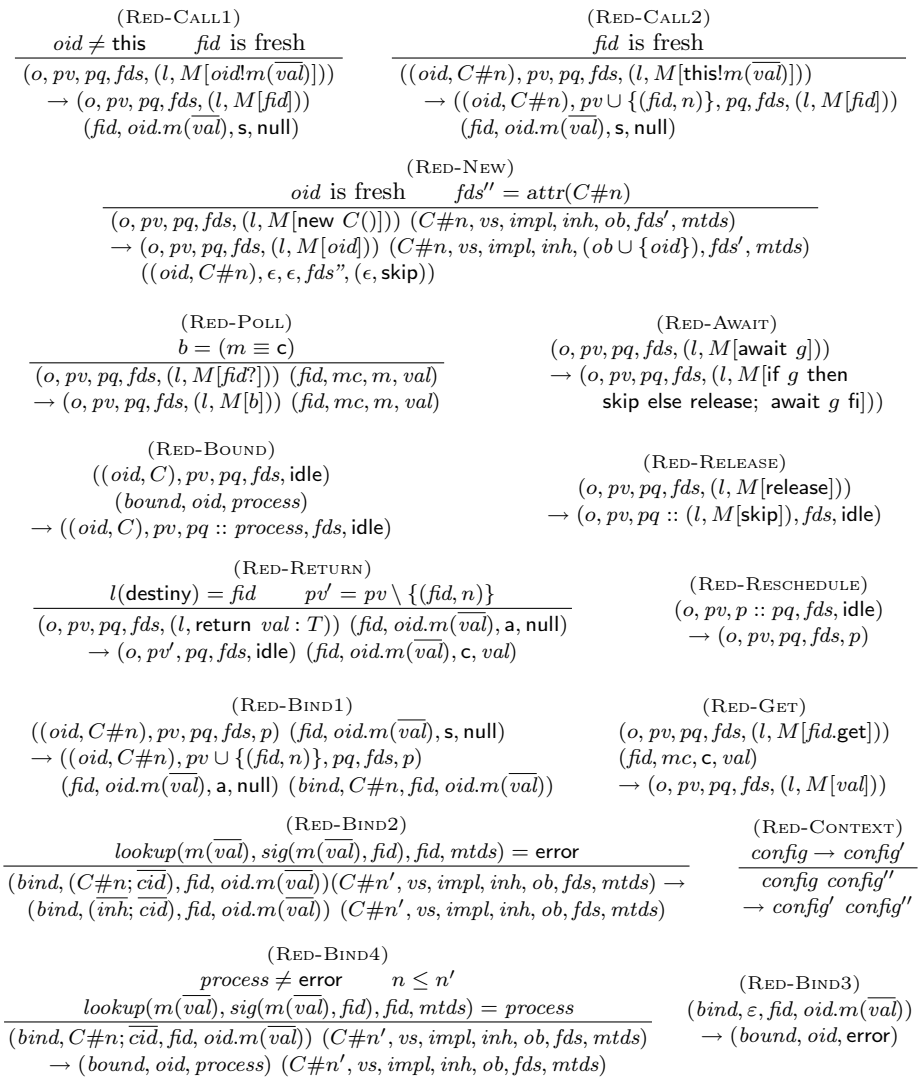

Fig. 4. The context reduction semantics.

Expressions and guards. In (Red-Call1) and (Red-Call2), external and internal asynchronous calls add a sleeping future to the configuration, returning its id to the caller. Note that an internal call extends the process version set with a pair consisting of the new future and the current stage number. This is because the asynchronous call to an internal method creates an obligation for the object to keep this method available until the call has been executed. In (Red-Get), a read on a future variable in the active process only reduces if the corresponding future is in completed mode. Otherwise, the process is blocked. In (Red-New), a new instance of a class C is introduced into the configuration (with fields

collected from C and its superclasses using $attr(C)$). In (Red-Poll), a future variable is polled to see if a call has been executed.

Release and rescheduling. Guards determine whether a process should be released. In (Red-Await), a process at a release point proceeds if its guard is true and releases otherwise. When a process is released, its guard is reused to reschedule the process. When an active process is released in (Red-Release) or terminates, it is replaced by the idle process, which allows a process from the process queue to be scheduled for execution in (Red-Reschedule).

Method invocation, binding, and return. A method call results in an activation on the callee's process queue. As the call is asynchronous, there is a delay between the call and its activation, represented by the sleeping mode of a future. After the call, (Red-Bind1) creates a bind request to the callee's class and the future changes its mode to active, preventing multiple activations. Note that the bind request extends the process version set with a pair consisting of the new future and the current stage number of the object, similar to an invocation for an internal call. The process version set influences the applicability of upgrades, delaying those upgrades that may introduce errors into the nonterminated processes. (Red-Bind2) traverses the implicit inheritance tree until binding fails in (Red-Bind3) or succeeds in (Red-Bind4). Successful binding results in a *bound* message to the callee, which is loaded into the process queue in (Red-Bound). When the process terminates, the result is stored by (Red-Return) in the future identified by the destiny variable. This future changes its mode to completed and the active process becomes idle. When a process terminates, its return value is placed in the associated future and the future id is removed from the process version set pv by (Red-Return). Finally, (Red-Context) reduces subconfigurations.

Adapting the type system to runtime configurations, we let $\Delta \vdash_R config$ ok denote that *config* is well-typed. The initial state of a well-typed program is well-typed, and type soundness can be established for the type system and reduction semantics of this paper (the details of the proof are given in [16]):

Theorem 1. *If $\Delta \vdash_R config$ ok and $config \to config'$, then there is an extension Δ' of Δ such that $\Delta' \vdash_R config'$ ok*

3 Example of Dynamic Class Extensions

Let an interface Account provide basic banking services; e.g., depositing money and receiving the balance for an account. Class BankAccount implements Account; an internal method increaseBalance is called by method deposit. The comment *V:0* indicates that this is class version 0.

```
class BankAccount implements Account {                    -- V:0
  Nat bal:=0;
  Bool increaseBalance (Nat sum) { bal := bal + sum; return true }
  Nat balance ( ) { return bal }
  Bool deposit (Nat sum) { return this!increaseBalance(sum) }}
```

By *dynamically extending* the class with new methods transfer and withdraw, money can be transferred to a receiver account or withdrawn. To log transactions,

a general method modifyBalance will modify the balance of the account and log the transaction:

```
class BankAccount implements Account {                                   -- V:1
  Nat bal:=0; Log l;
  Nat modifyBalance (Int sum) {Nat w; w := 0; l:= new Log(); bal := bal + sum;
    l.addlog(this, sum); if sum < 0 then w := -sum fi; return w }
  Bool increaseBalance(Nat sum) {bal := bal + sum; return true }
  Nat balance ( ) { return bal }
  Bool deposit (Nat sum) { fut⟨Nat⟩ w; w:=this!modifyBalance(sum); return true }
  Nat withdraw (Nat sum ) { await sum ≤ bal; return this!modifyBalance(-sum) }
  Bool transfer (Nat sum, Account acc) { fut⟨Nat⟩ w; await sum ≤ bal;
    w:=this!modifyBalance(-sum); return acc!deposit(w.get()) }}
```

Here, the class is extended with a new field l, new methods modifyBalance, withdraw, and transfer. Furthermore, deposit is redefined to use the internal method modifyBalance. (Remark that allowing asynchronous calls as statements in the language would remove the need for the future w in deposit.) However, the new methods withdraw and transfer are only known internally in the class. To *export* them the class is extended with a new interface TransferAcc with appropriate signatures for transfer and withdraw, after which transfer and withdraw may be invoked on pointers typed by TransferAcc. If we can type check that BankAccount implements TransferAcc, it is type-safe to bind a pointer typed by TransferAcc to an instance of BankAccount and call transfer and withdraw on this object:

```
class BankAccount implements Account, TransferAcc {                      -- V:2
  Nat bal:=0; Log l;
  Nat modifyBalance (Int sum) { Nat w; w := 0; l:= new Log(); bal := bal + sum;
    l.addlog(this, sum); if sum < 0 then w := -sum fi; return w }
  Bool increaseBalance(Nat sum) {bal := bal + sum; return true }
  Nat balance ( ) { return bal }
  Bool deposit (Nat sum) { fut⟨Nat⟩ w; w:=this!modifyBalance(sum); return true }
  Nat withdraw (Nat sum ) { await sum ≤ bal; return this!modifyBalance(-sum) }
  Bool transfer (Nat sum, Account acc) { fut⟨Nat⟩ w; await sum ≤ bal;
    w:=this!modifyBalance(-sum); return acc!deposit(w.get()) }}
```

As increaseBalance is now redundant, we *dynamically simplify* BankAccount by removing it. After the upgrades, the initial class definition has been replaced by

```
class BankAccount implements Account, TransferAcc {                      -- V:3
  Nat bal:=0; Log l;
  Nat modifyBalance (Int sum) { Nat w; w := 0; l:= new Log(); bal := bal + sum;
    l.addlog(this, sum); if sum < 0 then w := -sum fi; return w }
  Nat balance ( ) { return bal }
  Bool deposit (Nat sum) { fut⟨Nat⟩ w; w:=this!modifyBalance(sum); return true }
  Nat withdraw (Nat sum ) { await sum ≤ bal; return this!modifyBalance(-sum) }
  Bool transfer (Nat sum, Account acc) { fut⟨Nat⟩ w; await sum ≤ bal;
    w:=this!modifyBalance(-sum); return acc!deposit(w.get()) }}
```

These dynamic upgrades are here realized by three upgrade messages added to the running system: upgrading BankAccount with the redefinition of deposit and the new methods modifyBalance, withdraw and transfer; exporting new functionality by extending BankAccount with the TransferAcc interface; and removing the

redundant method increaseBalance. A type-safe introduction of these upgrades in a distributed system requires a combination of type checking and careful timing at runtime. Adding the new interface requires the presence of methods withdraw and transfer, so the first upgrade of BankAccount must already have occurred. Moreover, the class of l must implement Log. There are similar dependencies for removing increaseBalance: the redefinition of deposit must occur before the class simplification, otherwise method binding may fail. Similarly, we must ensure that the old definition of deposit is not a process in any runtime object.

4 Dynamic Classes

Software evolution in a running system may be perceived as a series of operations injected into the system, which modify the classes and the class hierarchy. An upgrade U is any dynamic class operation, as given by the following syntax:

$$\begin{array}{l} U ::= \mathsf{new\text{-}class}\ C\ \mathsf{extends}\ \overline{C}\ \mathsf{implements}\ \overline{I}\ \{\overline{T\ f};\overline{M}\} \quad | \ \mathsf{new\text{-}interface}\ I\ \mathsf{extends}\ \overline{I}\ \{\overline{M_s}\} \\ \quad | \ \mathsf{update}\ C\ \mathsf{extends}\ \overline{C}\ \mathsf{implements}\ \overline{I}\ \{\overline{T\ f};\overline{M}\} \quad | \ \mathsf{simplify}\ C\ \mathsf{retract}\ \overline{C}\ \{\overline{T\ f};\overline{M}\} \end{array}$$

A *class addition* adds the representation of the new class to the system, an *interface addition* extends the type system, a *class update* extends an existing class with new fields and methods and redefines existing methods in the class, and a *class simplification* removes redundant superclasses, fields, and methods from an existing class. Upgrades propagate asynchronously at runtime. They first change classes, then subclasses, and eventually the objects of those classes.

4.1 Typing of Dynamic Classes

Dynamic class operations are type checked in a sequence of typing environments $\Gamma^0, \Gamma^1, \ldots$, which extend each other; Γ^0 is the typing environment for the original program and Γ^i the current static view of the system. We describe the construction of Γ^{i+1} for the next well-typed upgrade U. The type system for judgments $\Gamma^{i+1} \vdash U$ is shown in Fig. 5, extending the system in Fig. 2. For simplicity, we omit the analysis of new-interface and focus on class updates. We assume that new interfaces are well-typed and that $\Gamma^i_{\mathcal{I}}$ are correctly extended for each update. (As before, we omit the straightforward analysis of superinterfaces and method signatures.) Rule (New-Class) for class additions requires a fresh name, type checks like a class in the original program, and extends $\Gamma^i_{\mathcal{C}}$. Remark that the version number of the new class is different from that of the program's original classes. This reflects the fact that the new class may depend on other dynamic changes to the system. For a new class, we remove the dependency to the class itself; i.e., $(C, 0)$ is removed from the dependency mapping.

Rule (Class-Extend) for the extension of a class C obeys a substitutability discipline captured by the predicate *refines*$(\overline{M}, \overline{M}_1)$; if $M \in \overline{M}$ redefines $M_1 \in \overline{M}_1$, the signature of M must be a subtype of the signature of M_1. The extended class replaces the definition of C in $\Gamma^i_{\mathcal{C}}$ by the binding Γ'. When retrieving the old version of the class from Γ^i_C, we represent the class compactly as a tuple

$$\frac{\begin{array}{c}\text{(New-Class)}\\ C \notin dom(\Gamma^i_{\mathcal{C}}) \quad \Gamma' = [C{\mapsto_C}(\overline{I}, \overline{C}, \overline{T\ f}, \overline{M})] \quad \forall I \in \overline{I} \cdot implements(C, I, \Gamma + \Gamma')\\ \forall M \in \overline{M} \ \cdot \ \Gamma^i + \Gamma' + [\mathsf{this}{\mapsto_v}C] + [attr(C, \Gamma^i + \Gamma')] \vdash M\,\langle \Sigma^M \rangle\end{array}}{\Gamma^i + \Gamma' + [\langle C, 1\rangle {\mapsto_d} \bigcup_{M \in \overline{M}} \Sigma^M \setminus \{(C,0)\}] \vdash \mathsf{new\text{-}class}\ C\ \mathsf{extends}\ \overline{C}\ \mathsf{implements}\ \overline{I}\ \{\overline{T\ f}; \overline{M}\}}$$

$$\frac{\begin{array}{c}\text{(Class-Extend)}\\ \Gamma^i_{\mathcal{C}}(C) = (\overline{I}_1, \overline{C}_1, \overline{T_1\ f_1}, \overline{M}_1) \quad \Gamma' = [C \mapsto_C (\overline{I}_1; \overline{I}, \overline{C}_1; \overline{C}, \overline{T_1\ f_1}; \overline{T\ f}, (\overline{M}_1 \oplus \overline{M}))]\\ vs = curr(C, \Gamma^i_d) \quad refines(\overline{M}, \overline{M}_1) \quad \forall I \in \overline{I} \cdot implements(C, I, \Gamma^i + \Gamma')\\ \forall M \in \overline{M} \ \cdot \ \Gamma^i + \Gamma' + [\mathsf{this}{\mapsto_v}C] + [attr(C, \Gamma^i + \Gamma')] \vdash M\,\langle \Sigma^M \rangle\end{array}}{\Gamma^i + \Gamma' + [\langle C, vs+1\rangle {\mapsto_d} \bigcup_{M \in \overline{M}} \Sigma^M \cup \{(C, vs)\}] \vdash \mathsf{update}\ C\ \mathsf{extends}\ \overline{C}\ \mathsf{implements}\ \overline{I}\ \{\overline{T\ f}; \overline{M}\}}$$

$$\frac{\begin{array}{c}\text{(Class-Simplify)}\\ \Gamma^i_{\mathcal{C}}(C) = (\overline{I}_1, \overline{C}_1, \overline{T_1\ f_1}, \overline{M}_1) \quad \Gamma' = [C \mapsto_C (\overline{I}_1, (\overline{C}_1 \setminus \overline{C}), (\overline{T_1\ f_1} \setminus \overline{T\ f}), (\overline{M}_1 \setminus \overline{M}))]\\ \overline{D} = \{C\} \cup below(C, \Gamma^i_{\mathcal{C}}) \} \quad dep = \textstyle\bigcup_{D \in \overline{D}} \{(D, curr(D, \Gamma^i_d))\} \quad vs = curr(C, \Gamma^i_d)\\ \forall D \in \overline{D} \cdot \Gamma^i + \Gamma' + [\mathsf{this}{\mapsto_v}D] + [attr(D, \Gamma^i + \Gamma')] \vdash (\Gamma^i_{\mathcal{C}} + \Gamma')(D).mtds\\ \forall D \in \overline{D} \wedge \ \forall I \in \Gamma^i_{\mathcal{C}}(D).impl \cdot implements(D, I, \Gamma^i + \Gamma')\end{array}}{\Gamma^i + \Gamma' + [\langle C, vs+1\rangle {\mapsto_d} dep \cup \{(C, vs)\}] \vdash \mathsf{simplify}\ C\ \mathsf{retract}\ \overline{C}\ \{\overline{T\ f}; \overline{M}\}}$$

Fig. 5. The type system for dynamic class extensions. Judgments have the form $\Gamma^{i+1} \vdash U$, where Γ^i is the current typing environment before the operation U.

and we denote by $\overline{M}_1 \oplus \overline{M}$ the union operation which retains methods in $\overline{M}$ in case of name conflicts. The function $curr(C, \Gamma^i_d)$ identifies the current version number of a class C by inspecting the dependency mapping. The new features of the class extension are type checked in a similar way as rule (Class) and the resulting dependencies, accumulated by the type analysis of methods, are bound to the new version $curr(C, \Gamma^i_d) + 1$ of the class in Γ^{i+1}_d. Moreover, in order to ensure that multiple upgrades to the same class occur in a correct order, the current version of the class is also included in this mapping.

In rule (Class-Simplify), which removes features from a class C, the simplification is restricted to superclasses, fields, and methods which are not statically needed in Γ^i. To verify this requirement, it is necessary to type check the new version of the class as well as its subclasses, identified by the function $below(C, \Gamma^i_{\mathcal{C}})$, in the updated typing environment Γ^{i+1}. The dependencies of the simplification operation are the current versions of the subclasses and of the class itself. As effects are not needed for the construction of the dependency mapping (the simplification only affects subclasses), for brevity, we elide the effects of methods and denote by $\Gamma(C).mtds$ and $\Gamma(C).impl$ the methods and interfaces of a class C and type check each single method and interface similar to rule (Class).

In the asynchronous setting of distributed concurrent objects, upgrades may be delayed and even bypass each other. Hence, the system reflected by the current typing environment Γ^i may differ considerably from the running system. To ensure that the execution is type safe, we exploit the dependency mapping of Γ^i to impose constraints on the applicability of the i'th upgrade at runtime. The constraints ensure that if one upgrade depends on another, they will be applied in the correct order, otherwise, they may be applied in any order, or in parallel.

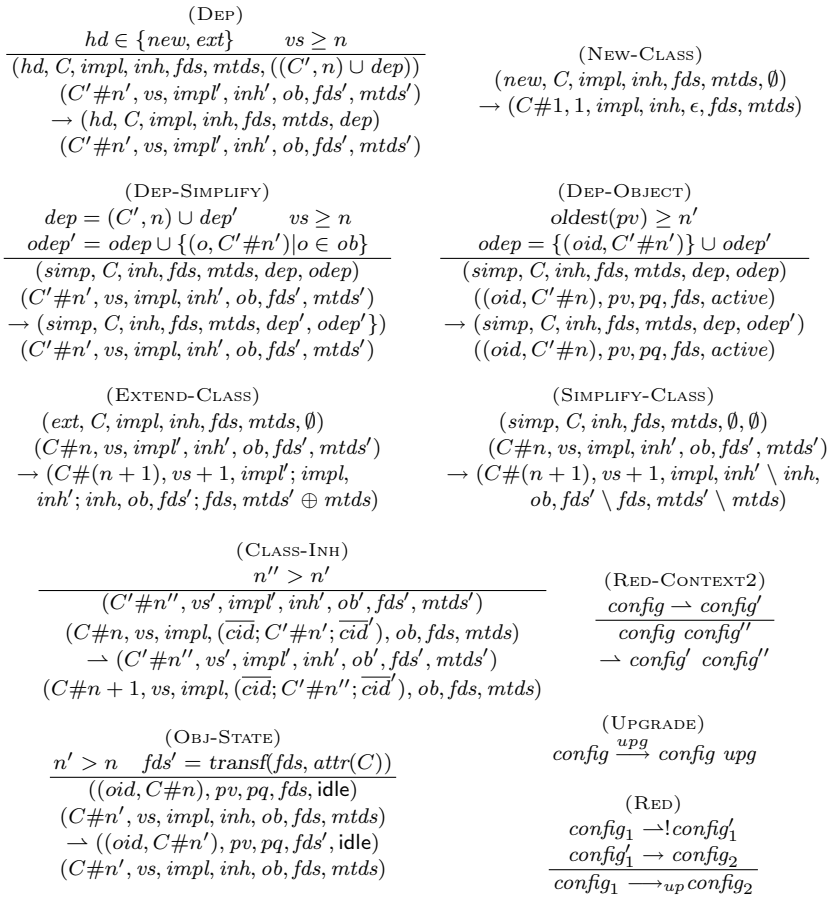

Fig. 6. The context reduction semantics for class upgrades.

4.2 Semantics for Dynamic Classes

We extend the runtime syntax of Figure 3 with upgrade messages as follows:

$$upg ::= (new, C, impl, inh, fds, mtds, dep) | (ext, C, impl, inh, fds, mtds, dep) \quad dep ::= \overline{(C, n)}$$
$$| \quad (simp, C, inh, fds, mtds, dep, odep) | \ldots \quad odep ::= \overline{(o, C\#n)}$$

An upgrade message for a new class or class extension has a class name C, a list *impl* of interfaces, a list *inh* of superclasses, a list *fds* of new fields, a set *mtds* of new (or redefined) methods, and a set *dep* of constraints to classes in the runtime system. For class simplification, *inh*, *fds* and *mtds* are the superclasses, fields and methods to be removed, respectively. For simplification, applicability not only depends on class constraints but also propagates to the state of runtime objects, as there may exist processes in or communication between objects in the runtime environment that uses fields or methods to be removed. Thus, in addition to class constraints, a class simplification message includes a set *odep* of constraints on

objects, which is initially empty but gradually extended during the verification of class constraints. If a message injected into the runtime configuration is well-typed in Γ^i, then *dep* is $\Gamma^i_d(\langle C, curr(C, \Gamma^i_d)\rangle)$. Thus, the static dependencies of the current upgrade are introduced into the runtime configuration.

The semantics for dynamic class operations extends the reduction system of Fig. 4 with the rules given in Fig. 6. A reduction step in the extended system takes the form $config_1 \longrightarrow_{up} config_2$ in (Red), where $config_1 \rightharpoonup! config'_1$ reduces $config_1$ to *normal form* by the relation $\rightharpoonup$, which consists of the two rules (Class-Inh) and (Obj-State), before the relation $\rightarrow$ applies. The $\rightharpoonup$ relation abstracts from locking disciplines that would otherwise be needed, as explained below.

Dynamic class operations are initiated by injecting a message *upg* into the configuration by (Upgrade). For the *extension* of a class C, this message is $(ext, C, impl, inh, fds, mtds, dep)$ which cannot be applied unless the constraints in *dep* are satisfied, checked by (Dep). Thus, the upgrade is delayed at runtime until other upgrades have been applied. When the constraints are satisfied, the superclasses, fields, and methods of the runtime class definition are extended and the stage and version numbers increased in (Extend-Class). (For the operator $\oplus$, see Sect. 4.1.) Similarly, (New-Class) creates a new runtime class when the constraints are satisfied. For the *simplification* of a class C, the message is $(simp, C, inh, fds, mtds, dep, \emptyset)$. When verifying class constraints in (Dep-Simplify), the set *ob* of instances of a class is used for stage constraints in *odep*. These are checked in (Dep-Object). To guarantee that an object is of stage n, we must ensure that all processes stemming from older versions of the class have completed and that there are no pending calls from such processes to local methods (which could be scheduled for removal). Rule (Dep-Object) compares stage constraints to the *oldest* stage number in the object's process version set *pv*. When no unsatisfied dependencies remain, the simplification can be applied in (Simplify-Class).

Updating the object state. When an object's class or superclass has been upgraded, the object's state must be updated *before* new code is allowed to execute. New instances of a class automatically get the new fields, but the upgrade of existing instances must be closely controlled; errors may occur if new or redefined methods, which rely on fields that are not yet available in the object, were executed. With recursive or nonterminating methods objects cannot generally be expected to reach a state without pending processes. Consequently, it is too restrictive to wait for the completion of all processes before applying an upgrade. However, objects may reach *quiescent* states when the processor has been released and before any pending process has been activated. Quiescent states are those in which the active process is idle. Any object which does not deadlock will eventually reach a quiescent state. In our language, nonterminating activity is defined by recursion, which ensures at least one quiescent state in each cycle.

Class upgrades propagate to objects in two steps. When a class C is upgraded in (Extend-Class) or (Simplify-Class), both its stage and version numbers increase. In order to notify objects of this change, the stage change propagates in rule (Class-Inh) to the subclasses of C, and the subclasses recursively increment their stage numbers. Since this notification is given priority, the object gets an upgrade

the next time it interacts with a class in rule (Red-Bind4). Before the new process is activated, the active process must become idle, in which case (Obj-State) applies. The *transf* function returns the new state, retaining the values of old fields.

The reduction $\longrightarrow_{up}$ in (Red) reduces a subconfiguration by $\rightharpoonup$ to its normal form before a $\rightarrow$ rewrite, simulating locking the object. Thus, the use of the $\rightharpoonup$ relation abstracts from two locking disciplines; one to deny access to classes for *bind* messages while (Class-Inh) is applicable and the other to delay processing *bind* messages from a callee by an upgraded class until the callee's state has been updated. A class may be upgraded several times before the object reaches a quiescent state, so the object may miss some upgrades. However a single state update suffices to ensure that the object, once upgraded, is a complete instance of the present version of its class. Extending Theorem 1, type soundness holds for the dynamic class system (the details of the proof are given in [16]):

Theorem 2 (Subject reduction). *Let P be a well-typed program with initial configuration init and let $U_1, \ldots, U_n$ be a series of well-typed dynamic class operations with runtime representation upg_i for U_i. If init $\longrightarrow_{up}$ config and upg_{i+1} is injected in the runtime configuration after upg_i for all $i < n$, then there is a typing context Δ such that $\Delta \vdash_R$ config* ok.

Proof (sketch). The proof is by induction over the number of reduction steps and then by cases. We show that injecting upg_i maintains the well-typedness of the configuration. Furthermore, we show that the dependencies provided by the static analysis enforce an ordering of upgrades such that new definitions give well-typed configurations. Especially, existing processes as well as new processes and fields in runtime objects are well-typed after the possible reductions.

5 Related Work

For many modern distributed applications, system availability during reconfiguration is crucial. Among dynamic or online upgrade solutions, version control systems aim at modular evolution; some keep multiple co-existing versions of a class or schema [3–5, 11, 13, 14], others apply a global update or "hot-swapping" [1, 6, 18, 19]. The approaches differ for active behavior, which may be disallowed [6, 13, 18, 19], delayed [1], or supported [14, 21]. Hjálmtýsson and Gray [14] propose proxy classes and reference indirection for C++, with multiple versions of each class. Old instances are not upgraded, so their activity is not interrupted. Existing approaches for Java, using proxies [19] or modifying the Java virtual machine [18], use global upgrade and do not apply to active objects.

Automatic upgrades by lazy global update has been proposed for distributed objects [1] and persistent object stores [6], in which instances of upgraded classes are upgraded, but inheritance and (nonterminating) active code are not addressed, limiting the effect and modularity of the class upgrade. Remark that the use of recursion instead of loops in our approach guarantees that all non-blocked processes will eventually reach a quiescent state. In [6] the ordering of upgrades is serialized and in [18] invalid upgrades raise exceptions.

It is interesting to apply formal techniques to systems for software evolution. Formalizations of runtime upgrade mechanisms are less studied, but exist for imperative [21], functional [4], and object-oriented [5] languages. In a recent upgrade system for (sequential) C [21], type-safe updates of type declarations and procedures may occur at annotated points identified by static analysis. However, the approach is synchronous as upgrades which cannot be applied immediately will fail. Closer to our work, UpgradeJ [5] uses an incremental type system in which class versions are only typechecked once. Our type system is incremental in this sense for new classes and class extensions, but class simplification requires rechecking the subclasses of the modified class. In contrast to our work, UpgradeJ is synchronous and uses explicit upgrade statements in programs. Upgrades only affect the class hierarchy and not running objects. Multiple versions of a class will coexist and the programmer must explicitly refer to the different class versions in the code. Compared to previous work by the authors [23], dynamic classes allow much more flexible runtime upgrades, including simplification operations which necessitate additional runtime overhead, yet the type system has been simplified. However, we do not support the removal of interfaces from classes. This would increase the runtime overhead significantly, as all objects with fields typed by that particular interface would need to be inspected.

6 Conclusion

This paper presents a kernel language for distributed concurrent objects in which programs may evolve at runtime by means of dynamic class operations. The granularity of this mechanism for reprogramming fits well with object orientation and it is modular as a single upgrade may affect a large number of running objects. The mechanism does not impose any particular requirements on the developers of initial applications and provides a fairly flexible mechanism for program evolution. It supports not only the addition of new interfaces and classes to running programs, but also the extension, redefinition, and simplification of existing classes. The dynamic class operations proposed in this paper integrate naturally with the concurrency model adapted in the Creol language and a prototype implementation has been integrated with Creol's execution platform.

The paper presents a type and effect system for dynamic class operations. A characteristic feature of our approach is that the type analysis identifies dependencies between different upgrades which are exploited at runtime to impose constraints on the runtime applicability of a particular upgrade in the asynchronous distributed setting, and suffice to guarantee type soundness. We currently investigate the application of other formal methods to dynamic class operations. In particular, lazy behavioral subtyping [9] seems applicable in order to extend the scope of object-oriented program logics to runtime program evolution.

References

1. S. Ajmani, B. Liskov, and L. Shrira. Modular software upgrades for distributed systems. In *Proc. ECOOP'06*, *LNCS* 4067, pages 452–476. Springer, 2006.

2. T. Amtoft, F. Nielson, and H. R. Nielson. *Type and Effect Systems: Behaviours for Concurrency*. Imperial College Press, 1999.
3. J. Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, 2007.
4. G. Bierman, M. Hicks, P. Sewell, and G. Stoyle. Formalizing dynamic software updating. In *Proc. 2nd Intl. Workshop on Unanticipated Software Evolution*, 2003.
5. G. Bierman, M. Parkinson, and J. Noble. UpgradeJ: Incremental typechecking for class upgrades. In *Proc. ECOOP'08*, *LNCS* 5142, pages 235–259. Springer, 2008.
6. C. Boyapati, B. Liskov, L. Shrira, C.-H. Moh, and S. Richman. Lazy modular upgrades in persistent object stores. In *Proc. OOPSLA'03*, pages 403–417. ACM Press, 2003.
7. D. Caromel and L. Henrio. *A Theory of Distributed Object*. Springer, 2005.
8. F. S. de Boer, D. Clarke, and E. B. Johnsen. A complete guide to the future. In *Proc. ESOP'07*, *LNCS* 4421, pages 316–330. Springer, Mar. 2007.
9. J. Dovland, E. B. Johnsen, O. Owe, and M. Steffen. Lazy behavioral subtyping. In *Proc. FM'08*, *LNCS* 5014, pages 52–67. Springer, May 2008.
10. S. Drossopoulou, F. Damiani, M. Dezani-Ciancaglini, and P. Giannini. More dynamic object re-classification: $\text{Fickle}_{\text{II}}$. *ACM TOPLAS*, 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. 6th Intl. Conf. on Functional Programming (ICFP'01)*, *ACM SIGPLAN notices* 36(10): 62–73, ACM Press.
12. M. Felleisen and R. Hieb. The revised report on the syntactic theories of sequential control and state. *Theor. Comp. Sci.*, 103(2):235–271, 1992.
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. 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.
15. A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM TOPLAS*, 23(3):396–450, 2001.
16. E. B. Johnsen, M. Kyas, and I. C. Yu. Dynamic classes: Modular asynchronous evolution of distributed concurrent objects. Research Report 383, Dept. of Informatics, Univ. of Oslo, Norway, May 2009.
17. E. B. Johnsen and O. Owe. An asynchronous communication model for distributed concurrent objects. *Software and Systems Modeling*, 6(1):35–58, Mar. 2007.
18. S. Malabarba, R. Pandey, J. Gragg, E. Barr, and J. F. Barnes. Runtime support for type-safe dynamic Java classes. In *Proc. ECOOP'00*, *LNCS* 1850, pages 337–361. Springer, June 2000.
19. A. Orso, A. Rao, and M. J. Harrold. A technique for dynamic updating of Java software. In *Proc. Intl. Conf. on Software Maintenance (ICSM'02)*, pages 649–658. IEEE Computer Society Press, Oct. 2002.
20. C. A. N. Soules *et al.* System support for online reconfiguration. In *Proc. USENIX Tech. Conf.*, pages 141–154, 2003.
21. G. Stoyle, M. Hicks, G. Bierman, P. Sewell, and I. Neamtiu. *Mutatis Mutandis*: Safe and predictable dynamic software updating. *ACM TOPLAS*, 29(4):22, 2007.
22. A. Welc, S. Jagannathan, and A. Hosking. Safe futures for Java. In *Proc. OOPSLA'05*, pages 439–453. ACM Press, 2005.
23. I. C. Yu, E. B. Johnsen, and O. Owe. Type-safe runtime class upgrades in Creol. In *Proc. FMOODS'06*, *LNCS* 4037, pages 202–217. Springer, June 2006.