Concurrent Objects à la Carte

Dave Clarke [1], Einar Broch Johnsen [2], and Olaf Owe [2]

[1] CWI, Amsterdam, the Netherlands
dave@cwi.nl
[2] Dept. of Informatics, University of Oslo, Norway
{einarj,olaf}@ifi.uio.no

Abstract Services are autonomous, self-describing, technology-neutral software units that can be described, published, discovered, and composed into software applications at run-time. Designing software services and composing services in order to form applications or composite services requires abstractions beyond those found in typical object-oriented programming languages. In this paper, we explore a number of the abstractions used in service-oriented computing and related Internet- and web-based programming models in the context of Creol, an executable concurrent object-oriented modeling language with active objects and futures; i.e., features capable of expressing and dealing with asynchronous actions. By adding various abstractions to the modeling language, we demonstrate how a concurrent object language may naturally address many of the requirements of service-oriented computing. The study of language extensions in the restricted setting of a small, high-level modeling language, such as Creol, suggests a cheap way of developing new abstractions for emerging application domains. In this paper, we explore abstractions in the context of service-oriented computing, particularly with regard to dynamic aspects such as service discovery and structuring mechanisms such as groups.

1 Introduction

Service-oriented computing is an emerging computational model in which services are autonomous, self-describing, technology-neutral software units that can be described, published, discovered, and composed into software applications at run-time. A service provides specific functionality to clients in its external environment. In particular, a client may detect new services or better service providers, and negotiate the use of services at run-time, depending on the quality and cost of the services currently available in the environment. This makes service-oriented computing an attractive model for distributed applications such as Web services, e-business, and e-government, but also for ad-hoc, self-configuring, and loosely-coupled networks, such as peer-to-peer networks, sensor-based applications, and mobile systems.

Research on service-oriented computing typically focuses on three aspects: service and data interchangeability (or platform-neutrality), technologies for service detection, and mechanisms for service composition and orchestration. *Interchangeability* is achieved through XML-related technologies which support the

exchange of structured or semi-structured data through a shared data model, as well as the publication and description of services. *Service discovery* (or open-endedness) is typically achieved through events and event-handling architectures, and via standards such as UDDI and service repositories. Industrial proposals for the description of *orchestration* mechanisms include BPML, WSFL, XLANG, and BPEL. These languages typically combine communication primitives with work-flow constructs in order to describe the distributed flow of control in flexible ways. Service-oriented computing has been formally studied through a plethora of process algebras (e.g., [8, 9, 24]), which mostly address aspects of orchestration. However, two approaches to service discovery in this context have recently been developed, extending process algebras with Linda tuple spaces to represent service repositories [9] and with semantic subtyping of channels [11].

Object orientation has been criticized for its apparent mismatch with service-oriented computing, partly due to a lack of advanced data types, which makes new solutions necessary to elegantly handle XML documents [6], and partly due to its traditionally restrictive communication and concurrency abstractions. The venue of truly parallel (distributed or multicore) systems challenges the multithread concurrency model which is predominant in object-oriented systems today and provided by Java and C#. This has lead to recent interest in concurrent objects and more flexible object interaction mechanisms [5, 10, 13, 18]. Concurrent objects are reminiscent of the concurrency model of Actors [2] and Erlang [3] and seem promising for deployment on multicore and distributed platforms, as well as for embedded systems (see Hooman and Verhoef [17]). Interestingly, concurrent objects have many of the properties described above for service-oriented computing. In particular, they interact via asynchronous communication and do not assume a tight coupling between caller and callee. This leads to a natural notion of asynchronous method call based on so-called futures [4, 22, 25, 29].

Creol is a concurrent object-oriented language which combines asynchronous method calls with controlled process release [18]. This decouples concurrency, communication, and synchronization in a very flexible way, while supporting a simple proof theory [13] compared to the proof theory of multithread concurrency [1]. The strict encapsulation of fields is enforced by using interfaces for typing objects; these interfaces include a so-called *cointerface* for callbacks between interacting objects. In this paper, we argue that this flexibility makes it easy to express orchestration patterns. Furthermore, we extend Creol with high-level primitives for service publishing and discovery, and for delegation and a hierarchical notion of service discovery based on groups. We give a type system and formal semantics for this extended language. Finally, we argue through a series of examples for its suitability for service-oriented computing.

The paper is structured as follows. Section 2 introduces Creol with primitives for service-oriented computing, its syntax and type system. Section 3 gives a reduction semantics for the language, in the style of Felleisen and Hieb [15]. Section 4 presents service-oriented computing flavored examples. Section 5 introduces abstractions for groups and their use through another example. Section 6 discusses our approach and some possible extensions in relation to existing work.

2 Service-Oriented Concurrent Objects

In this section, we present an extension of the concurrent object-oriented language Creol to address dynamic service publication and discovery. A concurrent object represents a separate computational unit, conceptually encapsulating its own processor. In Creol, method calls are asynchronous and object variables (references) are typed by interfaces. We use different interfaces to represent the different services offered by a concurrent object. In particular, an interface may require that a cointerface is offered by the service demander.

2.1 Syntax

The language syntax is given in Fig. 1. A program P is a list of interface and class definitions, followed by a method body corresponding to the main method. An interface D may inherit other interfaces and specify a set of method signatures. An interface is a subtype of all the interfaces it inherits. A class L may implement a set of interfaces and provide local fields and method definitions. For simplicity, we ignore class inheritance in this paper. We emphasize the differences with Java. As usual, a method signature declares the return type of the method, its name, and the types of its formal parameters. In addition, the method signature specifies the *cointerface* for the method; i.e., the required type of the object calling the method.

Expressions e are standard apart from the asynchronous method call $e!m(\overline{e})$, the (blocking) read operation $e.\mathbf{get}$, the asynchronous bind operation $\mathbf{bind}\ I$, and the service announcement operation $\mathbf{announce}\ I$. *Statements* s are standard apart from release points $\mathbf{await}\ g$ and $\mathbf{release}$, the delegation statement $\mathbf{forward}\ e!m(\overline{e})$ and the service removal statement $\mathbf{retract}\ e$. *Guards* g are conjunctions of Boolean expressions bv and polling operations $v?$ on futures v. When the guard in an $\mathbf{await}$ statement evaluates to $\mathbf{false}$, the processor is released and the active process is suspended, otherwise computation proceeds in the active process. A $\mathbf{release}$ statement suspends the active process and another suspended process may be rescheduled, similar to a yield in Java. (Each object has an associated processor, so locks, signaling or other forms of process control are not needed.) Non-deterministic choice $s \,\square\, s'$ allows either branch to be selected. The branches of a merge $s \parallel s'$ are interleaved at release points, influencing the control flow within a process without allowing other processes to execute, unless neither branch is enabled. In addition, the intermediate statement $s /\!/\!/ s'$ appears during reduction; it corresponds to the activation of statement s in the merge of statements s and s', where statement s' is delayed.

The sequence $t := e!m(\overline{e'})$; $\mathbf{await}\ t?$; $v := t.\mathbf{get}$, where t is a future variable, corresponds to a non-blocking method call, while $t := e!m(\overline{e'})$; $v := t.\mathbf{get}$ (or simply $v := e!m(\overline{e'}).\mathbf{get}$) corresponds to a blocking method call.

2.2 Typing

The typing rules are given in Fig. 2. Let Γ be a typing context which binds names to types (e.g., $x : T$) and write $\Gamma(x)$ for the type bound to x in Γ, which may

$$
\begin{array}{lll}
P & ::= \overline{D}\ \overline{L}\ \{\overline{T\ x}; s\} & D ::= \texttt{interface}\ I\ \texttt{extends}\ \overline{I}\ \{\overline{M_s}\} \\
M_s & ::= T\ m\ (\overline{T\ x})\ \texttt{with}\ I & L ::= \texttt{class}\ C\ \texttt{implements}\ \overline{I}\ \{\overline{T\ f}; \overline{M}\} \\
M & ::= M_s\{\overline{T\ x}; s\} & e ::= bv \mid \texttt{new}\ C(\,) \mid e.\texttt{get} \mid e!m(\overline{e}) \mid \texttt{null} \\
v & ::= f \mid x & \quad \mid \texttt{bind}\ I \mid \texttt{announce}\ I \\
b & ::= \texttt{true} \mid \texttt{false} & s ::= v := e \mid \texttt{await}\ g \mid \texttt{if}\ bv\ \texttt{then}\ s\ \texttt{else}\ s\ \texttt{fi} \\
bv & ::= b \mid v & \quad \mid \texttt{release} \mid \texttt{forward}\ e!m(\overline{e}) \mid \texttt{retract}\ e \\
T & ::= I \mid \texttt{bool} \mid \texttt{fut}(T) & \quad \mid s; s \mid s \,\square\, s \mid s \parallel s \mid s /\!/\!/ s \mid \texttt{return}\ e \mid \texttt{skip} \\
T^+ & ::= T \mid \texttt{ad} & g ::= bv \mid v? \mid g \wedge g
\end{array}
$$

Figure 1. The language syntax. Variables v are fields (f) or local variables (x), C is a class name, and I an interface name. Lists are indicated by overlining, as in $\overline{e}$. The type language is restricted to booleans, interfaces and futures. (Data types may be added.)

be undefined. For simplicity, we elide the checking of interface declarations, and assume that all methods declared in an interface have distinct names and that interface extensions are non-circular. Futures have explicit types; i.e., $\texttt{fut}(T)$ is the type for futures holding values of type T. In addition, the type **ad** is used to denote the type of service announcements and **ok** is the type of well-typed statements. Furthermore, if a class C implements interfaces $\overline{I}$, we conventionally use C as the name for the type $\texttt{interface}\ C\ \texttt{extends}\ \overline{I}\ \{\ \}$. The subtype relation $I \preceq J$ is induced by the reflexive and transitive closure of the interface extension relation. Data types as well as **ad** are only subtypes of themselves; i.e., $\texttt{bool} \preceq \texttt{bool}$, $\texttt{ad} \preceq \texttt{ad}$. In addition, if $T \preceq T'$ then $\texttt{fut}(T) \preceq \texttt{fut}(T')$.

In the typing rules for *expressions*, $\texttt{new}\ C$ gets type C, **null** can get any interface type, and $\texttt{announce}\ I$ gets type **ad**. We assume given an implicit table for interface declarations, so the auxiliary function $\mathit{lookup}(I, m, \overline{T})$ gives us the return and cointerface types of method m with formal parameters of types $\overline{T}$ in interface I, if such an m is declared in I. The typing of an asynchronous method call $e!m(\overline{e})$, where e has type I and $\overline{e}$ have types $\overline{T}$, can then be explained as follows: If method m is declared in the interface I of the callee e, with return type T and cointerface type J, then the asynchronous operation $e!m(\overline{e})$ is typed by $\texttt{fut}(T)$ if the caller **this** can provide the required cointerface. Similarly, the asynchronous operation $\texttt{bind}\ I$ is typed by $\texttt{fut}(I)$. Similarly, $e.\texttt{get}$ is of type T if e is a future of type $\texttt{fut}(T)$. In the typing of *guards*, the polling of a field f gets type **bool** provided that f is a future. For the typing of *statements*, note that the delegation statement $\texttt{forward}\ e!m(\overline{e})$ is well-typed if the caller of the current method could have called $e!m(\overline{e})$ instead of the call to the current method. It follows in typing rule FORWARD that $\texttt{forward}\ e!m(\overline{e})$ is well-typed provided that m has an adequate return type and that the current caller has a type which satisfies the cointerface of m. The service revocation statement $\texttt{retract}\ v$ is well-typed if v has type **ad**. Note that rule ASSIGN allows expressions of type T^+ (in contrast to, e.g., CALL). The typing of the remaining statements is standard. For the typing of methods, note that the typing context is extended with the two variables **destiny**, which provides the type of the method call's associated future, and **caller**, which is typed by the method's cointerface and

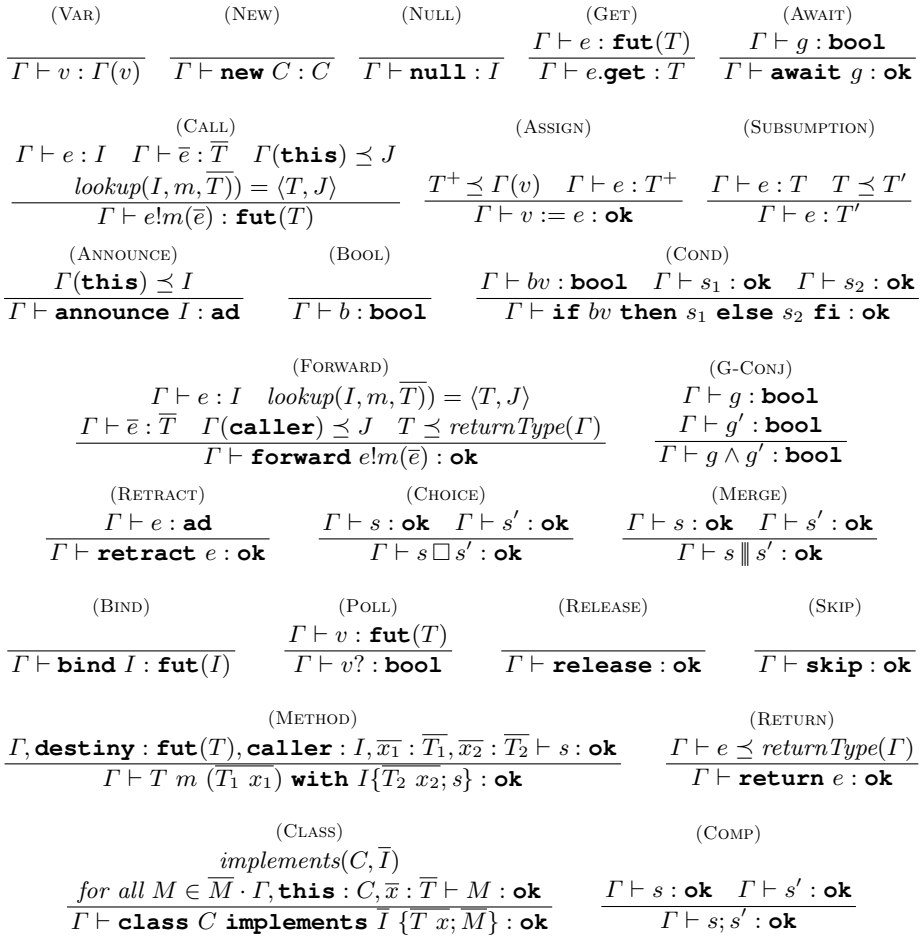

Figure 2. The typing rules. The predicate $implements(C, \overline{I})$ compares the method signatures of C with those of each interface I in $\overline{I}$ with respect to co- and contravariance requirements for formal parameters and cointerfaces. Function $returnType(\Gamma) = T$ whenever $\Gamma(\mathtt{destiny}) = \mathtt{fut}(T)$.

provides support for callback, in addition to types for formal parameters and locally declared variables. As usual, the typing rule for classes extends the typing environment with a type for **this** and types for the fields declared in the class. In addition, rule Class includes a check for compatibility with the declared interfaces $\overline{I}$ of the class C by means of an auxiliary predicate $implements(C, \overline{I})$ which ensures that for all method signatures $T_1\ m\ (\overline{T_2}\ x)$ **with** A declared in an interface I or a superinterface of I (for $I \in \overline{I}$), there is a method declaration $T_3\ m\ (\overline{T_4}\ x)$ **with** $B\ \{\overline{T\ y}; s\}$ in C such that $T_3 \preceq T_1$, $\overline{T_2} \preceq \overline{T_1}$, and $A \preceq B$.

$$
\begin{array}{rcl@{\qquad}rcl}
\mathit{config} & ::= & \epsilon \mid \mathit{object} \mid \mathit{msg} \mid \mathit{service} \mid \mathit{config}\ \mathit{config} & \mathit{fds} & ::= & \overline{f\ d} \\
\mathit{object} & ::= & (\texttt{obj}\ \mathit{oid}, C, \mathit{processQ}, \mathit{fds}, \mathit{active}) & \mathit{active} & ::= & \mathit{process} \mid \texttt{idle} \\
\mathit{processQ} & ::= & \epsilon \mid \mathit{process} \mid \mathit{processQ}\ \mathit{processQ} & \mathit{service} & ::= & (\texttt{ad}\ \mathit{adid}, I, \mathit{oid}) \\
\mathit{msg} & ::= & (\texttt{fut}\ \mathit{fid}, \mathit{cmd}, \mathit{oid}, \mathit{mode}, d) & \mathit{process} & ::= & (\overline{T\ x\ d}, s) \\
d & ::= & \mathit{oid} \mid \mathit{fid} \mid \mathit{adid} \mid \texttt{null} \mid b & \mathit{cmd} & ::= & \mathit{oid}!m(\overline{d}) \mid \texttt{bind}\ I
\end{array}
$$

Figure 3. Syntax for runtime configurations. Here, d denotes data, including Boolean values (b) and identifiers for objects (oid), futures (fid), and service announcements ($adid$). Processes include code and local state, i.e. local variables with types and values.

3 Operational Semantics

The semantics is a small-step reduction relation on *configurations* of objects, messages, and services (see Fig. 3). An *object* has an identifier, a class, a queue of suspended processes, fields, and an active process. The process **idle** indicates that no method is running in the object. We introduce the notion of a command, *cmd*, to model asynchronous actions. Commands are reduced using a single rule which spawns off a new thread, in effect, to execute the command. Initially, commands include asynchronous method calls and bind requests. A *future* (**fut** *fid*, *cmd*, *oid*, *mode*, *d*) captures the state of command: initially *sleeping*, the command may later becomes *active*, and finally, when *completed*, it stores its result in the future. The value $mode \in \{\texttt{s}, \texttt{a}, \texttt{c}\}$ represents these three states. A bind request does not use the mode **a**. Default values for types are given by a function *default* (e.g., $default(I) = \texttt{null}$, $default(\texttt{bool}) = \texttt{false}$, and $default(\texttt{fut}(T)) = \texttt{null}$). The *initial configuration* of a program $\overline{L}\ \{\overline{T\ x}; s\}$ contains one object $(\texttt{obj}\ o, \emptyset, \epsilon, (\overline{T\ x\ default(T)}, s))$.

Reduction takes the form of a relation $config \to config'$. Rules apply to partial configurations and may be applied in parallel. This differs from the semantics of object-oriented languages with a global store [16], but is consistent with Creol's [19] executable Maude semantics [12] and allows true concurrency in the distributed or multicore setting. The main rules are given in Figs. 4, 5, and 6. The context reduction semantics decomposes a statement into a reduction context and a redex, and reduces the redex [15]. *Reduction contexts* are statements S, expressions E, and guards G with a single hole denoted by $\bullet$:

$$
\begin{array}{rcl}
S & ::= & \bullet \mid v := E \mid S; s \mid S /\!\!/\!\!/ s \mid \texttt{if}\ G\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \texttt{fi} \\
 & & \mid \texttt{return}\ E \mid \texttt{forward}\ E \mid \texttt{retract}\ E \mid \texttt{await}\ G \\
E & ::= & \bullet \mid E.\texttt{get} \mid E!m(\overline{e}) \mid \mathit{oid}!m(\overline{d}, E, \overline{e}) \\
G & ::= & \bullet \mid E? \mid G \wedge g \mid b \wedge G
\end{array}
$$

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

$$
\begin{array}{rcl}
\mathit{stat\text{-}redexes} & ::= & x := d \mid f := d \mid \texttt{await}\ g \mid \texttt{skip}; s \mid \texttt{if}\ b\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \texttt{fi} \\
 & & \mid \texttt{release} \mid \texttt{forward}\ \mathit{oid}!m(\overline{d}) \mid \texttt{retract}\ \mathit{aid} \mid \texttt{return}\ d \\
\mathit{expr\text{-}redexes} & ::= & x \mid f \mid \mathit{fid}.\texttt{get} \mid \mathit{oid}!m(\overline{d}) \mid \texttt{new}\ C() \mid \texttt{bind}\ I \mid \texttt{announce}\ I \\
\mathit{guard\text{-}redexes} & ::= & \mathit{fid}? \mid b \wedge g
\end{array}
$$

(Red-Cmd)
$$\frac{\textit{fid} \text{ is fresh}}{(\texttt{obj}\ \textit{oid}, C, \textit{pq}, \textit{fds}, (l, S[\textit{cmd}])) \to (\texttt{obj}\ \textit{oid}, C, \textit{pq}, \textit{fds}, (l, S[\textit{fid}]))\ (\texttt{fut}\ \textit{fid}, \textit{cmd}, \textit{oid}, \texttt{s}, \texttt{null})}$$

(Red-Get)
$$(\texttt{obj}\ \textit{oid}, C, \textit{pq}, \textit{fds}, (l, S[\textit{fid}.\texttt{get}]))\ (\texttt{fut}\ \textit{fid}, \textit{cmd}, \textit{oid}', \texttt{c}, d) \to (\texttt{obj}\ \textit{oid}, C, \textit{pq}, \textit{fds}, (l, S[d]))\ (\texttt{fut}\ \textit{fid}, \textit{cmd}, \textit{oid}', \texttt{c}, d)$$

(Red-New)
$$\frac{\textit{oid}' \text{ is fresh} \quad \textit{fds}' = \textit{defaults}(C')}{(\texttt{obj}\ \textit{oid}, C, \textit{pq}, \textit{fds}, (l, S[\texttt{new}\ C'()])) \to (\texttt{obj}\ \textit{oid}, C, \textit{pq}, \textit{fds}, (l, S[\textit{oid}']))(\texttt{obj}\ \textit{oid}', C', \epsilon, \textit{fds}', \textit{proc}(C, \textit{run}, \textit{null}, \textit{oid}', \varepsilon))}$$

(Red-Poll)
$$\frac{b = (\textit{mode} \equiv \texttt{c})}{(\texttt{obj}\ \textit{oid}, C, \textit{pq}, \textit{fds}, (l, S[\textit{fid}?]))\ (\texttt{fut}\ \textit{fid}, \textit{cmd}, \textit{oid}', \textit{mode}, d) \to (\texttt{obj}\ \textit{oid}, C, \textit{pq}, \textit{fds}, (l, S[b]))\ (\texttt{fut}\ \textit{fid}, \textit{cmd}, \textit{oid}', \textit{mode}, d)}$$

(Red-Await)
$$(\texttt{obj}\ \textit{oid}, C, \textit{pq}, \textit{fds}, (l, S[\texttt{await}\ g])) \to (\texttt{obj}\ \textit{oid}, C, \textit{pq}, \textit{fds}, (l, S[\texttt{if}\ g\ \texttt{then skip else release};\ \texttt{await}\ g\ \texttt{fi}]))$$

(Red-Release)
$$\frac{S[\texttt{release}] \neq S'[\texttt{release}; s /\!\!/\!\!/ s']}{(\texttt{obj}\ \textit{oid}, C, \textit{pq}, \textit{fds}, (l, S[\texttt{release}])) \to (\texttt{obj}\ \textit{oid}, C, \textit{pq} :: (l, S[\texttt{skip}]), \textit{fds}, \texttt{idle})}$$

(Red-Reschedule)
$$(\texttt{obj}\ \textit{oid}, C, \textit{pq} :: p :: \textit{pq}', \textit{fds}, \texttt{idle}) \to (\texttt{obj}\ \textit{oid}, C, \textit{pq} :: \textit{pq}', \textit{fds}, p)$$

Figure 4. The context reduction semantics (1/4). In the last rule, pq and pq' may be empty. In Red-New, the *proc* function gives an activation of *run* (with oid' as caller).

Filling the hole of a context S with an expression r is denoted $S[r]$.

Expressions and guards. Red-Cmd spawns asynchronous actions by adding a sleeping future to the configuration, returning its identifier to the caller. In Red-Get, a read on a future variable blocks the active process until the future is in **c**ompleted mode (since no other rule applies to this redex). Blocking does not reschedule a suspended process. Object creation in Red-New introduces a new instance of a class C into the configuration (with fields collected from C), and the *run* method is called. In Red-Poll, a future variable is polled to see if the asynchronous action has completed.

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 otherwise releases. 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. The rule given

(Red-Method-Bind)

$$(\mathbf{obj}\ oid, C, pq, fds, p)\ (\mathbf{fut}\ fid, oid!m(\overline{d}), oid', \mathbf{s}, \mathbf{null})$$
$$\rightarrow (\mathbf{obj}\ oid, C, pq :: proc(C, m, fid, oid', \overline{d}), fds, p)\ (\mathbf{fut}\ fid, oid!m(\overline{d}), oid', \mathbf{a}, \mathbf{null})$$

(Red-Return)

$$\frac{l(\mathbf{destiny}) = fid}{(\mathbf{obj}\ oid, C, pq, fds, (l, S[\mathbf{return}\ d]))\ (\mathbf{fut}\ fid, oid''!m(\overline{d}), oid', \mathbf{a}, \mathbf{null}) \rightarrow (\mathbf{obj}\ oid, C, pq, fds, \mathbf{idle})\ (\mathbf{fut}\ fid, oid''!m(\overline{d}), oid', \mathbf{c}, d)}$$

(Red-Forward)

$$\frac{l(\mathbf{destiny}) = fid \quad l(\mathbf{caller}) = oid'}{(\mathbf{obj}\ oid, C, pq, fds, (l, S[\mathbf{forward}\ oid''!m(\overline{d})]))\ (\mathbf{fut}\ fid, \_, \_, \_, \_) \rightarrow (\mathbf{obj}\ oid, C, pq, fds, \mathbf{idle})\ (\mathbf{fut}\ fid, oid''!m(\overline{d}), oid', \mathbf{s}, \mathbf{null})}$$

(Red-Announce)

$$\frac{adid \text{ is fresh}}{(\mathbf{obj}\ oid, C, pq, fds, (l, S[\mathbf{announce}\ I])) \rightarrow (\mathbf{obj}\ oid, C, pq, fds, (l, S[adid]))\ (\mathbf{ad}\ adid, I, oid))}$$

(Red-Retract)

$$(\mathbf{obj}\ oid, C, pq, fds, (l, S[\mathbf{retract}\ adid]))\ (\mathbf{ad}\ adid, I, oid) \rightarrow (\mathbf{obj}\ oid, C, pq, fds, (l, S[\mathbf{skip}]))$$

(Red-Service-Bind)

$$\frac{I' \preceq I}{(\mathbf{fut}\ fid, \mathbf{bind}\ I, o, \mathbf{s}, \mathbf{null})\ (\mathbf{ad}\ adid, I', oid) \rightarrow (\mathbf{fut}\ fid, \mathbf{bind}\ I, o, \mathbf{c}, oid)\ (\mathbf{ad}\ adid, I', oid)}$$

Figure 5. The context reduction semantics (2/4). The function *proc* returns the process corresponding to method m of class C (or a superclass) with default values for the local variables, and **destiny**, **caller**, and the formal parameters, initialized with the actual parameter values (here, *fid*, oid', and $\overline{d}$, respectively).

here is nondeterministic with respect to which process to choose. Active behavior is initiated by a special method, named *run*, which is activated when the object is created. The interleaving of active and reactive behavior is controlled by means of release points.

Method invocation, return, and delegation. 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-Method-Bind creates a new process reflecting the activation of the called method, in which the **destiny** variable stores the return address to the call's future. Thus when the process terminates, the result is stored by Red-Return in the future identified by **destiny**. This future changes its mode to **c**ompleted and the active process becomes **idle**. The delegation of a call to a method of another object in Red-Forward modifies the future associated with the current call; the new method call is inserted and the mode of the future is reset to **s**, but the *fid* and caller values of the previous call are kept.

(Red-Choice1)
$$\frac{\mathit{enabled}(s, (\mathit{fds}, l), \mu)}{(\mathbf{obj}\ \mathit{oid}, C, \mathit{pq}, \mathit{fds}, (l, S[s \,\square\, s'])) \ \mu \to (\mathbf{obj}\ \mathit{oid}, C, \mathit{pq}, \mathit{fds}, (l, S[s])) \ \mu}$$

(Red-Choice2)
$$\frac{\mathit{enabled}(s', (\mathit{fds}, l), \mu)}{(\mathbf{obj}\ \mathit{oid}, C, \mathit{pq}, \mathit{fds}, (l, S[s \,\square\, s'])) \ \mu \to (\mathbf{obj}\ \mathit{oid}, C, \mathit{pq}, \mathit{fds}, (l, S[s'])) \ \mu}$$

(Red-Merge1)
$$(\mathbf{obj}\ \mathit{oid}, C, \mathit{pq}, \mathit{fds}, (l, S[s \parallel s'])) \to (\mathbf{obj}\ \mathit{oid}, C, \mathit{pq}, \mathit{fds}, (l, S[s /\!/\!/ s']))$$

(Red-Merge2)
$$(\mathbf{obj}\ \mathit{oid}, C, \mathit{pq}, \mathit{fds}, (l, S[s \parallel s'])) \to (\mathbf{obj}\ \mathit{oid}, C, \mathit{pq}, \mathit{fds}, (l, S[s' /\!/\!/ s]))$$

(Red-Merge-Skip)
$$(\mathbf{obj}\ \mathit{oid}, C, \mathit{pq}, \mathit{fds}, (l, S[\mathbf{skip} /\!/\!/ s])) \to (\mathbf{obj}\ \mathit{oid}, C, \mathit{pq}, \mathit{fds}, (l, S[s]))$$

(Red-Merge-Release1)
$$\frac{\mathit{enabled}(s', (\mathit{fds}, l), \mu)}{(\mathbf{obj}\ \mathit{oid}, C, \mathit{pq}, \mathit{fds}, (l, S[\mathbf{release}; s /\!/\!/ s'])) \ \mu \to (\mathbf{obj}\ \mathit{oid}, C, \mathit{pq}, \mathit{fds}, (l, S[s' /\!/\!/ s])) \ \mu}$$

(Red-Merge-Release2)
$$\frac{\neg \mathit{enabled}(s', (\mathit{fds}, l), \mu)}{(\mathbf{obj}\ \mathit{oid}, C, \mathit{pq}, \mathit{fds}, (l, S[\mathbf{release}; s /\!/\!/ s'])) \ \mu \to (\mathbf{obj}\ \mathit{oid}, C, \mathit{pq}, \mathit{fds}, (l, S[\mathbf{release}; (s \parallel s')])) \ \mu}$$

(Red-Var-Local)
$$(\mathbf{obj}\ \mathit{oid}, C, \mathit{pq}, \mathit{fds}, (l, S[x])) \to (\mathbf{obj}\ \mathit{oid}, C, \mathit{pq}, \mathit{fds}, (l, S[l(x)]))$$

(Red-Assign-Local)
$$(\mathbf{obj}\ \mathit{oid}, C, \mathit{pq}, \mathit{fds}, (l, S[x := d])) \to (\mathbf{obj}\ \mathit{oid}, C, \mathit{pq}, \mathit{fds}, (l[x \mapsto d], S[\mathbf{skip}]))$$

(Red-Field)
$$(\mathbf{obj}\ \mathit{oid}, C, \mathit{pq}, \mathit{fds}, (l, S[f])) \to (\mathbf{obj}\ \mathit{oid}, C, \mathit{pq}, \mathit{fds}, (l, S[\mathit{fds}(f)]))$$

(Red-Assign-Field)
$$(\mathbf{obj}\ \mathit{oid}, C, \mathit{pq}, \mathit{fds}, (l, S[f := d])) \to (\mathbf{obj}\ \mathit{oid}, C, \mathit{pq}, \mathit{fds}[f \mapsto d], (l, S[\mathbf{skip}]))$$

(Red-Skip)
$$(\mathbf{obj}\ \mathit{oid}, C, \mathit{pq}, \mathit{fds}, (l, S[\mathbf{skip}; s])) \to (\mathbf{obj}\ \mathit{oid}, C, \mathit{pq}, \mathit{fds}, (l, S[s]))$$

(Red-Cond1)
$$(\mathbf{obj}\ \mathit{oid}, C, \mathit{pq}, \mathit{fds}, (l, S[\mathbf{if\ true\ then}\ s_1\ \mathbf{else}\ s_2\ \mathbf{fi}])) \to (\mathbf{obj}\ \mathit{oid}, C, \mathit{pq}, \mathit{fds}, (l, S[s_1]))$$

(Red-Cond2)
$$(\mathbf{obj}\ \mathit{oid}, C, \mathit{pq}, \mathit{fds}, (l, S[\mathbf{if\ false\ then}\ s_1\ \mathbf{else}\ s_2\ \mathbf{fi}])) \to (\mathbf{obj}\ \mathit{oid}, C, \mathit{pq}, \mathit{fds}, (l, S[s_2]))$$

(Red-Context)
$$\frac{\mathit{config} \to \mathit{config}'}{\mathit{config}\ \mathit{config}'' \to \mathit{config}'\ \mathit{config}''}$$

(Red-empty)
$$(\mathbf{obj}\ \mathit{oid}, C, \mathit{pq}, \mathit{fds}, (l, \mathbf{skip})) \to (\mathbf{obj}\ \mathit{oid}, C, \mathit{pq}, \mathit{fds}, \mathbf{idle})$$

(Red-Parallel)
$$\frac{\mathit{config}\ \mu \to \mathit{config}'\ \mu' \quad \mathit{config}''\ \mu \to \mathit{config}'''\ \mu'' \quad \mathrm{dom}(\mu) = \mathrm{dom}(\mu') = \mathrm{dom}(\mu'') \quad \mathrm{dom}(\mathit{config}') \cap \mathrm{dom}(\mathit{config}''') = \emptyset}{\mathit{config}\ \mathit{config}''\ \mu \to \mathit{config}'\ \mathit{config}'''\ \mu' \odot \mu''}$$

Figure 6. The context reduction semantics (3/4). Here, μ denotes a configuration of futures. The rule Red-Merge-Release2 assumes that μ includes all futures in the configuration. The notation $\sigma[v \mapsto d]$ is used to update the binding of v in a state σ. The obvious rule for conjunction is omitted. The function dom(*config*) extracts the set of identities of the objects, services, and futures in the configuration *config*.

Service discovery. In Red-Announce, an object makes public that it provides a given service, and in Red-Retract an object removes a service announcement.

If a service has been announced by some object, the discovery of that service is done by initiating an asynchronous command **bind** I via RED-CMD. This request is later bound to an object providing the service in RED-SERVICE-BIND, which stores the identity of an object announcing the service in the future.

Choice, merge, and release. Either branch of a choice or merge may be selected for reduction, captured by RED-CHOICE1, RED-CHOICE2, RED-MERGE1, and RED-MERGE2. When a branch of a merge statement completes, RED-MERGE-SKIP schedules the other branch. If a release occurs inside a merge, the other branch of the merge is the first candidate for rescheduling — rescheduling is local to a process whenever possible. If both branches release, then the process is released. Let σ map fields and local variables to their values. Process release is based on the predicate *enabled* defined on guards, futures, and states which determines whether a guard will not directly release:

$$\begin{aligned} \mathit{enabled}(b, \sigma, \mu) &= b \\ \mathit{enabled}(v, \sigma, \mu) &= \mathit{enabled}(\sigma(v), \sigma, \mu) \\ \mathit{enabled}(\mathit{fid}?, \sigma, \mu) &= \mathit{mode} \equiv \mathtt{c}, \text{ where } (\mathit{fid}, \_, \_, \mathit{mode}, \_) \in \mu \\ \mathit{enabled}(g \wedge g', \sigma, \mu) &= \mathit{enabled}(g, \sigma, \mu) \wedge \mathit{enabled}(g', \sigma, \mu) \end{aligned}$$

The predicate is lifted to statements; $\mathit{enabled}(\mathtt{await}\ g, \sigma, \mu) = \mathit{enabled}(g, \sigma, \mu)$ is the crucial case. In RED-MERGE-RELEASE1, RED-MERGE-RELEASE2 and RED-RELEASE, the contexts and redexes do not factor expressions involving **release** uniquely: these may be factored as both $S[\mathtt{release}]$ and $S'[\mathtt{release}; s /\!\!/\!\!/ s']$. A clause is added to RED-RELEASE to ensure that $\mathtt{release}; s /\!\!/\!\!/ s'$ is preferred.

Context and parallel reductions. A reduction applies to a subconfiguration by rule RED-CONTEXT. In RED-PARALLEL futures may be shared between parallel reductions. As the futures inspected by one process may be changed by another, they need to be recomposed in a consistent way. This is handled by a function $\mu \odot \mu'$ which collects futures from μ and μ' and resolves conflicting futures with the same *fid* (i.e., keeping the completed ones) (See [13]). New futures are located in *config*$'$ and *config*$'''$.

Sample reduction. The following is a short reduction sequence illustrating how **bind** works, in combination with the **get** operation for accessing a future. The overline indicates the expression/command being evaluated. The underline indicates the context in which this evaluation occurs.

$$\begin{aligned} & \underline{(\mathtt{obj}\ \mathit{oid}, C, \mathit{pq}, \mathit{fds}, (l, (\overline{\mathtt{bind}\ I}).\mathtt{get}))}\ (\mathtt{ad}\ \mathit{aid}, I', \mathit{oid}') \\ \to\ & (\mathtt{obj}\ \mathit{oid}, C, \mathit{pq}, \mathit{fds}, (l, \mathit{fid}.\mathtt{get}))\ \underline{(\mathtt{fut}\ \mathit{fid}, \overline{\mathtt{bind}\ I}, \mathit{oid}, \mathtt{s}, \mathtt{null})}\ (\mathtt{ad}\ \mathit{aid}, I', \mathit{oid}') \\ \to\ & \underline{(\mathtt{obj}\ \mathit{oid}, C, \mathit{pq}, \mathit{fds}, (l, \overline{\mathit{fid}.\mathtt{get}}))\ (\mathtt{fut}\ \mathit{fid}, \mathtt{bind}\ I, \mathit{oid}, \mathtt{c}, \mathit{oid}')}\ (\mathtt{ad}\ \mathit{aid}, I', \mathit{oid}') \\ \to\ & (\mathtt{obj}\ \mathit{oid}, C, \mathit{pq}, \mathit{fds}, (l, \mathit{oid}'))\ (\mathtt{fut}\ \mathit{fid}, \mathtt{bind}\ I, \mathit{oid}, \mathtt{c}, \mathit{oid}')\ (\mathtt{ad}\ \mathit{aid}, I', \mathit{oid}') \end{aligned}$$

4 Examples

We first consider how Creol's high-level concurrency structuring primitives can be applied to orchestration patterns in wide-area computing, by adapting some

examples from Misra and Cook [24]. We then show how the proposed constructs for service discovery and delegation may be used in the context of a restaurant example. To simplify the notation in the examples, we allow expressions $\texttt{o!m}(\overline{e})$ as program statements if m has return type `void` in the interface of o (and omit the return statement in the body of such methods). Furthermore, we let `Any` be a minimal interface implemented by all classes (thus, the supertype of all other interfaces) and omit specifying the cointerface if it is `Any`.

Example 1. Consider services which provide news on request. These may be modeled by a class `NewsSite` offering an interface `News` to the environment with a method `news`, which for a given date returns a document with the news of that day from a "site". Let `CNN` and `BBC` be two such sites; i.e., two variables typed by the `News` interface. By calling $\texttt{CNN!news}(d)$ or $\texttt{BBC!news}(d)$ the news for a specified date d will be (asynchronously) downloaded in `XML` format from that site. (We do not here consider how XML can be integrated as a data type in Creol, see [28].) Another site distributes emails to clients at request. This site may be modeled by a class `EmailSite` which implements an interface `Email` with a method $\texttt{send}(m, a)$, where m is some message content in `XML` format and a is the address of a `Client` object. Let `NewsService` be an interface for sites combining news from different sites, as defined by the following class:

```
class NewsServiceSite implements NewsService {
  News CNN; Email email;

  void run() { CNN := new News(); email := new Email() }
  bool newsRelay(Date d, Client a) {
    XML v, fut(XML) t;
    t:= CNN!news(d); await t?; v:= t.get; email!send(v,a);
    return true
  }
}
```

In this class, a method relays news from `CNN` for a given date `d` to a given client `a` by invoking the `send` service of the `email` object with the result returned from the call to `CNN!news`, if `CNN` responds. For simplicity, the method returns **`true`** upon termination. Note that a `NewsServiceSite` object executing a `newsRelay` process is not blocked while waiting for the `CNN` or the `email` object to respond. Instead the process is suspended until the response has arrived. Once the object responds, the process becomes enabled and may resume its execution, receiving the news and eventually forwarding the news by email. If either of these objects never responds the `NewsServiceSite` object may proceed with its other activity, only the suspended process will never become enabled.

Now consider the case where a client wants news from both objects `CNN` and `BBC`. Because there may be significant delays until an object responds, the client desires to have the news from each object relayed whenever it arrives. This is naturally modeled by the merge operator, as in the following method:

```
  bool newsRelay2(Date d, Client a) with Any {
    XML v; fut(XML) t1; fut(XML) t2;
```

```
    t1 := CNN!news(d); t2 := BBC!news(d);
     (await t1?; v:= t1.get; email!send(v,a))
       ||| (await t2?; v:= t2.get; email!send(v,a))
}
```

The merge operator allows the news pages from BBC and CNN to be forwarded to the email!send service once they are received. If neither service responds, the whole process is suspended and can be activated again when a response is received. By executing the method, at most two emails are sent.

If the first news page available from either CNN or BBC is desired, the nondeterministic choice operator may be used to invoke the email!send service with the result received from either CNN or BBC, as in the following method:

```
bool newsRelay3(Date d, Client a) {
  XML v; fut(XML) t1; fut(XML) t2;
    t1 := CNN!news(d); t2 := BBC!news(d);
      ((await t1?; v:= t1.get) □ (await t2?; v:= t2.get));
       email!send(v,a)
}
```

Here news is requested from both news objects, but only the first response will be relayed. The latest arriving response is ignored.

Example 2. At a restaurant, the host of a dinner party is in charge of ordering wine. For this purpose, the host needs to attract the attention of some passing waiter. If busy, the waiter delegates the order to another waiter. The waiter who eventually deals with the order needs to get a more precise specification of the wine by querying the host; e.g., the desired vintage. The waiter then brings the new bottle to the host of the dinner party. We model this scenario by two interfaces Sommelier and Customer where the latter is the cointerface of the former, thus enabling callback to the customer. We shall conventionally annotate the interface itself with the cointerface information; in this case, the same cointerface applies to all methods declared in that interface.

```
interface Customer with Waiter {
  int whichVintage()
}
interface Sommelier with Customer {
  bool orderWine(string producer)
}
```

The implementation is given in Fig. 7 and demonstrates the dynamic aspects of the proposed primitives for service-oriented computing. We let a class Host implement the interface Customer and another class Waiter implement Sommelier. Waiters initially publish their interface, and retract this service if they have too many customers. Hosts dynamically find waiters when they need to make orders. When too busy, a waiter dynamically finds another waiter and forwards the order. Furthermore, the Waiter class demonstrates active behavior, represented by management of service announcements given in the run method, interleaved with reactive behavior, represented by the orderWine

```
class Host implements Customer, HostDuties {
 bool enoughWine;

 void run() { this!entertainGuests()
 }
 void surveyWineBottle() { Waiter waiter; fut(bool) ack;
  await not(enoughWine); waiter := (bind Waiter).get;
  ack := waiter!orderWine("MyFavoriteWine");
  await ack?; enoughWine := true; release; // enjoy the wine
  this!entertainGuests()
 }
 void entertainGuests() {
   enoughWine := false; this!surveyWineBottle()
 }
 int whichVintage() { return 1970 }
}

class Waiter(int maxLoad) implements Sommelier {
 int noOfCustomers;

 void run() { ad service;
  await noOfCustomers < maxLoad;
  service := announce Waiter;
  await noOfCustomers ≥ maxLoad;
  retract service; this!run()
 }
 bool orderWine(string producer) with Customer {
   fut(int) order; int vintage; Sommelier colleague;
   if (noOfCustomers ≥ maxLoad) then
    colleague := bind Waiter.get;
    forward colleague!orderWine(producer)
   else noOfCustomers := noOfCustomers + 1;
    order := caller!whichWine();
    await order?; vintage := order.get; // serve bottle
    noOfCustomers := noOfCustomers - 1; return true fi
 }
}
```

Figure 7. The restaurant example. In class `Waiter`, the `maxLoad` attribute is declared as a class parameter, to be instantiated upon object creation.

method. The `Host` class additionally implements an interface `HostDuties`, which specifies methods `entertainGuests` and `surveyWineBottle`. Note that in the `orderWine` method of the `Waiter` class, the callback to `whichWine` is type correct since `Customer` is the cointerface.

5 Groups

Groups [7,21,23] provide a way of structuring advertisements and hence objects into virtual organisations. For example, groups are use in the JXTA peer-to-peer library [20] to structure peers, advertisements, and channels in order to represent the parties involved in (il)legally downloading a particular film. Groups also offer the possibility of providing access control to service discovery, such as restricting members of the group to only those with legal right to the film. For simplicity, we consider only a very rudimentary access control mechanism, whereby an object has to be a member of a group in order to interact with it. Technically, the operations for service discovery and announcement and their semantics (Sections 2 and 3), will be generalised to the scope of particular groups.

Groups may be created and dissolved, and objects may join and leave groups. Finally, advertisement and discovery happens in the context of a group. For uniformity we denote by **glob** the top-level group in which to advertise and search for advertisements by default. Syntactically groups are treated like objects, and operations on groups like asynchronous method calls. In order to enable groups to be advertised, along with objects, we need to extend the notion of announcement. In Section 2, announcements were purely *subjective*; i.e., an object can only announce itself. In contrast *objective* announcements have the form $e!\mathbf{announce}(e', I)$, where the result of expression e' is advertised using interface I—again assuming that e' has type I—into the group resulting from expression e. In order to discover a service advertised in a group, the object must be a member of that group.

Syntax. The syntax introduced in Fig. 1 is extended as follows. Let **group** be the type of group identifiers, ranged over by *gid*, and let I include **group** in order to transparently enable the advertising of groups. Let the *expressions* e include the following operations on groups: **new group**, $e!\mathbf{announce}(e', I)$, and $e!\mathbf{bind}(I)$. Thus, the previous advertisement and discovery operations are redundant: subjective, groupless announcements **announce** I may be encoded as $\mathbf{glob}!\mathbf{announce}(\mathbf{this}, I)$ and $\mathbf{bind}(I)$ as $\mathbf{glob}!\mathbf{bind}(I)$. Finally, let the *statements* s include the following operations on groups: $e!\mathbf{dissolve}$, $e!\mathbf{join}$, and $e!\mathbf{leave}$. All operations involving groups are asynchronous.

Typing. Fig. 8 presents the type rules for the new operations, which simply ensure that these operations apply to groups. Rule Announce-in-Group requires that the announced value conforms with the announced type. Rule Bind-in-Group is the obvious extension to the rule Bind of Fig. 2.

Runtime Syntax. The runtime syntax introduced in Fig. 3 is extended as follows. Let *id* be either *oid* or *gid*. We extend the notion of command as follows:

$$\begin{aligned} cmd ::= \ldots \mid \mathbf{new\ group} \mid gid!\mathbf{dissolve} \mid gid!\mathbf{join} \\ \mid \ gid!\mathbf{leave} \mid gid!\mathbf{announce}(id, I) \mid gid!\mathbf{bind}(I) \end{aligned}$$

Configurations *config* are extended with a representation $(\mathbf{gr}\ gid, oids, aids)$ for a group with identifier *gid*, whose members are objects represented by the set

$$\frac{}{\Gamma \vdash \texttt{new group} : \texttt{fut}(\texttt{group})}\ (\textsc{Group-Create}) \qquad \frac{\Gamma \vdash e : \texttt{group}}{\Gamma \vdash e!\texttt{dissolve} : \texttt{ok}}\ (\textsc{Group-Dissolve}) \qquad \frac{\Gamma \vdash e : \texttt{group}}{\Gamma \vdash e!\texttt{join} : \texttt{ok}}\ (\textsc{Group-Join})$$

$$\frac{\Gamma \vdash e : \texttt{group}}{\Gamma \vdash e!\texttt{leave} : \texttt{ok}}\ (\textsc{Group-Leave}) \qquad \frac{\Gamma \vdash e : I \quad \Gamma \vdash e' : \texttt{group}}{\Gamma \vdash e'!\texttt{announce}(e, I) : \texttt{fut}(\texttt{ad})}\ (\textsc{Announce-in-Group}) \qquad \frac{\Gamma \vdash e : \texttt{group}}{\Gamma \vdash e!\texttt{bind}(I) : \texttt{fut}(I)}\ (\textsc{Bind-in-Group})$$

Figure 8. Typing for group-related operations

oids and containing the advertisements *aids*. The reduction contexts become:

$$\begin{aligned} S ::= \ldots \mid\ & E!\texttt{dissolve} \mid E!\texttt{join} \mid E!\texttt{leave} \\ \mid\ & E!\texttt{announce}(e, I) \mid d!\texttt{announce}(E, I) \mid E!\texttt{bind}(I) \end{aligned}$$

The redexes become:

$$\begin{aligned} \textit{stat-redexes} &::= \ldots \mid \textit{gid}!\texttt{dissolve} \mid \textit{gid}!\texttt{join} \mid \textit{gid}!\texttt{leave} \\ \textit{expr-redexes} &::= \ldots \mid \textit{gid}!\texttt{bind}(I) \mid \texttt{new group} \mid \textit{gid}!\texttt{announce}(\textit{id}, I) \end{aligned}$$

Reduction Rules. In Fig. 5, Red-Cmd applies to the asynchronous evaluation of commands. The reduction rules for the new operations are given in Fig. 9. As all new operations are asynchronous, the reduction rules operate in the context of some future data structure, generally depending upon the fact that the caller is stored within this data structure in order to perform a primitive access control check. Apart from Red-Create, Red-Join and Red-Leave an object needs to be a member of the group in order to interact with it. Rule Red-Create creates a new group which contains no advertisements or members and Red-Dissolve removes an entire group. Rule Red-Join and Red-Leave allow an object to join and leave a group, by modifying the list of group members. Rule Red-Announce creates a new announcement and places it in the group and Red-Retract removes an announcement from a group, generalising the earlier version of this rule. Rule Red-Bind binds to an announcement in the scope of a given group.

Example 3. Groups may be used to structure the services of auction houses. The underlying services are managed by the `ItemAuction` class, corresponding to an item for sale in an auction. The items are grouped into auctions, presumably the collection of items to be auctioned during a particular day or perhaps of a specific kind of item (e.g., guitars). Auctions are organised into auction houses, which again are groups.

To simplify the presentation, variable initialization may occur in declarations, with the syntax $T\ v = e$. Furthermore, we use a simple form of multicast; i.e., for each object in $\overline{e}$, $\overline{e}!m(\overline{e'})$ produces one future and $e'!\texttt{announce}(\overline{e}, I)$ makes announcements, adding the objects to the group e'. Some basic functions on the data type `Set[T]` are assumed to be defined, such as *add* and *remove*.

The following class illustrates how some auctions could be created and populated from an external source.

(Red-Create)
$$\frac{gid \text{ is fresh}}{(\textbf{fut}\ fid, \textbf{new group}, oid, \textbf{s}, \textbf{null}) \rightarrow (\textbf{fut}\ fid, \textbf{new group}, oid, \textbf{c}, gid)\ (\textbf{gr}\ gid, \epsilon, \epsilon)}$$

(Red-Dissolve)
$$\frac{oid \in oids}{(\textbf{fut}\ fid, gid!\textbf{dissolve}, oid, \textbf{s}, \textbf{null})\ (\textbf{gr}\ gid, \_, oids) \rightarrow (\textbf{fut}\ fid, gid!\textbf{dissolve}, oid, \textbf{c}, \textbf{null})}$$

(Red-Join)
$$(\textbf{fut}\ fid, gid!\textbf{join}, oid, \textbf{s}, \textbf{null})\ (\textbf{gr}\ gid, oids, aids) \rightarrow (\textbf{fut}\ fid, gid!\textbf{join}, oid, \textbf{c}, \textbf{null})\ (\textbf{gr}\ gid, oids \cup \{oid\}, aids)$$

(Red-Leave)
$$(\textbf{fut}\ fid, gid!\textbf{leave}, oid, \textbf{s}, \textbf{null})\ (\textbf{gr}\ gid, oids, aids) \rightarrow (\textbf{fut}\ fid, gid!\textbf{leave}, oid, \textbf{c}, \textbf{null})\ (\textbf{gr}\ gid, oids \setminus \{oid\}, aids)$$

(Red-Announce)
$$\frac{oid \in oids \quad aid \text{ is fresh}}{(\textbf{fut}\ fid, gid!\textbf{announce}(id, I), oid, \textbf{s}, \textbf{null})\ (\textbf{gr}\ gid, oids, aids) \rightarrow (\textbf{fut}\ fid, gid!\textbf{announce}\ (id, I), oid, \textbf{c}, \textbf{aid})\ (\textbf{ad}\ aid, I, id)\ (\textbf{gr}\ gid, oids, aids \cup \{aid\})}$$

(Red-Retract)
$$\frac{oid \in oids \quad aid \in aids}{(\textbf{fut}\ fid, \textbf{retract}\ aid, oid, \textbf{s}, \textbf{null})\ (\textbf{ad}\ aid, I, id)\ (\textbf{gr}\ gid, oids, aids) \rightarrow (\textbf{fut}\ fid, \textbf{retract}\ aid, oid, \textbf{c}, \textbf{null})\ (\textbf{gr}\ gid, oids, aids \setminus \{aid\})}$$

(Red-Bind)
$$\frac{oid \in oids \quad aid \in aids}{(\textbf{fut}\ fid, gid!\textbf{bind}(I), oid, \textbf{s}, \textbf{null})\ (\textbf{ad}\ aid, I, id)\ (\textbf{gr}\ gid, oids, aids) \rightarrow (\textbf{fut}\ fid, gid!\textbf{bind}(I), oid, \textbf{c}, id)\ (\textbf{ad}\ aid, I, id)\ (\textbf{gr}\ gid, oids, aids)}$$

Figure 9. The context reduction semantics (4/4) for group-related operations.

```
class Main(ItemDatabase db) {
  void run() {
    group ah1 = (new group).get;
                 this!populate("Christie's", ah1);
    group ah2 = (new group).get;
                 this!populate("Dave's", ah2);
    group as3 = (new group).get;
                 this!populate("Einar and Olaf's", ah3);
  }
  void populate(String name, group house) {
    Set[Item] items;
    group auction = (new group).get;
    house!join; house!announce(auction, group);
```

```
    // populate auction from external source
    items := db!itemsFor(name).get;
    auction!announce(items, Item) // multicast
   }
}
```

The class `ItemAuction` manages the auction of an individual item, including a set of the bidders for that item, and notifications to those bidders when the auction status changes. The method `bid` handles individual bids, relying on the mutual exclusion provided by Creol objects, and notifies other bidders. Class `ItemAuction` implements an interface `Item` with `Bidder` as cointerface and methods `showInterest`, `loseInterest`, and `bid`. (The `Bidder` interface and its implementation are given further below.)

```
class ItemAuction implements Item {
 int currentBid = 0;
 Bidder holder = null;
 bool sold = false;
 Set[Bidder] bidders = new Set();

 bool bid(int amount) with Bidder {
  if (sold or amount <= current) then return false fi;
  bidders := add(bidders, caller);
  currentBid := amount; holder := caller;
  remove(bidders,caller)!higherBid(amount); // multicast
  return true
 }

 // allow a bidder to (un)subscribe to an item auction
 void showInterest() with Bidder {
  bidders := add(bidders,caller)
 }
 void loseInterest() with Bidder {
  bidders := remove(bidders,caller)
 }

 void run() { await currentBid > 0; this!watchBids() }

 void watchBids() {
   int previousBid = currentBid;
   // going once, twice, three times:
   release; release; release;
   if currentBid > previousBid then this!watchBids()
   else sold := true; // accept current bid
    holder!success();
    bidders!auctionClosed() fi // multicast
 }
}
```

Specific items can be implemented by extending `Item`, giving a more specific description (in lieu of meta-data).

```
interface Guitar extends Item {}
interface Gibson extends Guitar {}
interface Firebird extends Gibson {}
interface FirebirdI extends Firebird {}
interface NonReverseFirebird extends Firebird {}
```

`Bidder` is an interface implemented by potential buyers in order to receive notifications from the auctions they follow. The notifications include that a higher bid has been made, that the auction has closed, and that a bid was successful. In each case, the **`caller`** is the item to which the notifications refer.

```
interface Bidder with Item {
  void higherBid(int bid);
  void auctionClosed();
  void success()
}
```

`Buyers` use service discovery to find groups denoting auction houses, and within an auction house, an auction will be found, within which the specific items of interest can be found.

```
class Buyer implements Bidder {
  void run() {
    // find an auction house -- needs meta-data
    group ah = glob!bind(group).get; ah!join;
    group auction = ah!bind(group).get; auction!join;
    Item item = auction!bind(Firebird).get;
    item!showInterest()
  }
  // Implementation of the Bidder interface
  void higherBid(int bid) with Item {
     if (bid < myMax) then caller!bid(bid + 10) fi
  }
  void auctionClosed() with Item {...} // quit, go home sad
  void success() with Item {...} // quit, go home happy
}
```

6 Discussion

This paper explored a number of abstractions for service-oriented computing and related internet- and web-based programming models in the context of the concurrent object-oriented (modeling) language Creol. In particular, we have considered abstractions of dynamic aspects such as service discovery and structuring mechanisms such as groups. By adding abstractions to the modeling language, we showed how Creol satisfies many of the requirements of service-oriented computing. Directly introducing abstractions into the language and implementing them in Maude [12], is a quick way of enabling model development for emerging application domains, simultaneously providing a rapid way of evaluating alternative design decisions. There are many ways to extend Creol to enhance its ability to accurately model real world systems. We now explore a few of these.

Open system and failure models New services can come on-line at any time. This can be modeled by external interaction with a running Creol expression. External interaction may be captured by a labelled transition—and the existing set of rewrite rules would need to be modified to propagate such transitions:

$$\epsilon \xrightarrow{\texttt{inject}\ \mathit{object}} \mathit{object} \quad \text{(Meta-Inject)}$$

The initiation of Creol's class update mechanism [30] may be modeled this way, and may be used to provide runtime upgrade of services.

Dually, it is possible to model when services go off-line, either temporarily or permanently. The following rule considers an explicit **remove** command:

$$(\texttt{fut}\ \mathit{fid}, \texttt{remove}\ \mathit{oid}, \mathit{oid}', \texttt{s}, \_)\ (\texttt{obj}\ \mathit{oid}, C, \mathit{processQ}, \mathit{fds}, \mathit{active}) \\ \rightarrow (\texttt{fut}\ \mathit{fid}, \texttt{remove}\ \mathit{oid}, \mathit{oid}', \texttt{c}, \texttt{null}) \quad \text{(Red-Remove)}$$

A similar meta-rule interacting with the environment could model the operation.

In general, operations in distributed systems can only be considered as partial due to (intermittent) failures of the network. A partial solution in the context of Creol is to modify asynchronous calls to allow the possibility of timing-out. A uniform way to achieve this is to add time-outs to future objects, as follows:

$$(\texttt{fut}\ \mathit{fid}, \mathit{cmd}, \mathit{caller}, \underline{\underline{\mathit{timer}}}, \mathit{mode}, \mathit{result}),$$

where *timer* has one of the following values **timer infinity** or **timer** n, for $n \geq 0$. An additional mode **to** is introduced to model the occurrence of a time-out. The new, under-the-hood, reduction rules are:

$$\texttt{timer}\ (n+1) \rightarrow \texttt{timer}\ n \quad \text{(Red-Timer)}$$

$$\frac{\mathit{status} \neq \texttt{c}}{(\texttt{fut}\ \mathit{fid}, \mathit{cmd}, \mathit{caller}, \texttt{timer}\ 0, \mathit{status}, \mathit{result}) \\ \rightarrow (\texttt{fut}\ \mathit{fid}, \mathit{cmd}, \mathit{caller}, \texttt{timer}\ 0, \texttt{to}, \mathit{result})} \quad \text{(Red-Timeout)}$$

Changes at the language level, such as the interaction between **get** and **await** and time-outs would need to be done, perhaps via an exception mechanism. Maude has a real-time extension [26] to help implement these features.

Enhancing Groups Although we used subinterfaces (of `Guitar`) to enable more precise service discovery, group discovery was rather weak, as there were no means for distinguishing one group from another. More precise groups can be specified by applying inheritance to groups, as follows:

```
group QualityGroup;
group GuitarGroup;
group GibsonGroup extends GuitarGroup, QualityGroup;
```

Group discovery would be based on a supergroup; e.g., `QualityGroup`. Group creation would then be extended to create a specific kind of group:

```
new GibsonGroup;
```

Further possibilities include limiting the types that can be advertised in a group:

```
group GuitarGroup of Guitar;
```

Additional meta-data such as tags and certificates (for access control) [20] could be supplied to enhance the quality of service discovery.

Additional group operations would certainly increase Creol's modeling power; e.g., broadcast to a group [21]. Likewise, a stream-based approach to asynchronous responses, as in JXTA [20], to model multiple potential candidate bindings, would enable a client to better select the most appropriate service.

Enhancing the security model underlying our groups would make them more appropriate for modeling real-world applications. Such measures are necessary since one would not generally want everyone to be able to dissolve the group, and some members may be able to advertise, whereas others may not.

Other directions for future work include adding behavioural types (e.g., session types [14]) to place more static control on the use of objects or services, and to add a coordination layer (channels, connectors, or tuple space(s) [27]) between the objects or services, rather than allowing direct communication.

Acknowledgment

The authors are indebted to Willem-Paul de Roever for encouragement, inspiration, and interaction over a number of years. In particular, he has given substantial support to the Creol and Credo projects, which form the basis of this paper. We are also grateful to Frank de Boer for interesting discussions on service binding in the context of object orientation.

References

1. E. Ábrahám-Mumm, F. S. de Boer, W.-P. de Roever, and M. Steffen. Verification for Java's reentrant multithreading concept. In *Intl. Conf. on Foundations of Software Science and Computation Structures (FOSSACS'02)*, volume 2303 of *Lecture Notes in Computer Science*, pages 5–20. Springer, Apr. 2002.
2. G. A. Agha. *ACTORS: A Model of Concurrent Computations in Distributed Systems*. The MIT Press, Cambridge, Mass., 1986.
3. J. Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, 2007.
4. H. G. Baker and C. E. Hewitt. The incremental garbage collection of processes. *ACM SIGPLAN Notices*, 12(8):55–59, 1977.
5. N. Benton, L. Cardelli, and C. Fournet. Modern concurrency abstractions for C♯. *ACM Transactions on Programming Languages and Systems*, 26(5):769–804, Sept. 2004.

6. G. M. Bierman, E. Meijer, and W. Schulte. The essence of data access in Cω. In A. P. Black, editor, *Proc. 19th Eur. Conf. on Object-Oriented Programming (ECOOP'05)*, volume 3586 of *Lecture Notes in Computer Science*, pages 287–311. Springer, 2005.
7. K. P. Birman. The process group approach to reliable distributed computing. *Communications of the ACM*, Dec. 1993.
8. M. Boreale, R. Bruni, L. Caires, R. D. Nicola, I. Lanese, M. Loreti, F. Martins, U. Montanari, A. Ravara, D. Sangiorgi, V. T. Vasconcelos, and G. Zavattaro. SCC: A service centered calculus. In M. Bravetti, M. Núñez, and G. Zavattaro, editors, *Proc. 3rd Intl. Workshop on Web Services and Formal Methods*, volume 4184 of *Lecture Notes in Computer Science*, pages 38–57. Springer, Sept. 2006.
9. M. Bravetti and G. Zavattaro. Service oriented computing from a process algebraic perspective. *Journal of Logic and Algebraic Programming*, 70(1):3–14, 2007.
10. D. Caromel and L. Henrio. *A Theory of Distributed Object*. Springer, 2005.
11. G. Castagna, R. D. Nicola, and D. Varacca. Semantic subtyping for the pi-calculus. In *Proc. 20th IEEE Symp. on Logic in Computer Science (LICS'05)*, pages 92–101. IEEE Computer Society Press, 2005.
12. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada. Maude: Specification and programming in rewriting logic. *Theoretical Computer Science*, 285:187–243, Aug. 2002.
13. F. S. de Boer, D. Clarke, and E. B. Johnsen. A complete guide to the future. In R. de Nicola, editor, *Proc. 16th Eur. Symp. on Programming (ESOP'07)*, volume 4421 of *Lecture Notes in Computer Science*, pages 316–330. Springer, Mar. 2007.
14. M. Dezani-Ciancaglini, D. Mostrous, N. Yoshida, and S. Drossopoulou. Session types for object-oriented languages. In D. Thomas, editor, *Proc. 20th Eur. Conf. on Object-Oriented Programming (ECOOP'06)*, volume 4067 of *Lecture Notes in Computer Science*, pages 328–352. Springer, 2006.
15. M. Felleisen and R. Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 103(2):235–271, 1992.
16. M. Flatt, S. Krishnamurthi, and M. Felleisen. A programmer's reduction semantics for classes and mixins. In *Formal Syntax and Semantics of Java*, volume 1523 of *Lecture Notes in Computer Science*, pages 241–269. Springer, 1999.
17. J. Hooman and M. Verhoef. Formal semantics of a VDM extension for distributed embedded systems. In D. Dams, U. Hannemann, and M. Steffen, editors, *Correctness, Concurrency, and Components. Festschrift for Willem-Paul de Roever*, Lecture Notes in Computer Science (this volume). Springer, 2008.
18. E. B. Johnsen and O. Owe. An asynchronous communication model for distributed concurrent objects. *Software and Systems Modeling*, 6(1):35–58, Mar. 2007.
19. E. B. Johnsen, O. Owe, and I. C. Yu. Creol: A type-safe object-oriented model for distributed concurrent systems. *Theoretical Computer Science*, 365(1–2):23–66, Nov. 2006.
20. JXTA Project. `http://jxta.org`.
21. D. Lea. Objects in groups. Available on the web: `gee.cs.oswego.edu/dl/groups/groups.html`, Dec. 1993.
22. B. H. Liskov and L. Shrira. Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems. In D. S. Wise, editor, *Proc. SIGPLAN Conf. on Programming Language Design and Implementation (PLDI'88)*, pages 260–267. ACM Press, June 1988.
23. S. Maffeis. Electra: Making distributed programs object-oriented. In *Proc. USENIX Symp. on Experiences with Distributed and Multiprocessor Systems*, Sept. 1993.

24. J. Misra and W. R. Cook. Computation orchestration: A basis for wide-area computing. *Software and Systems Modeling*, 6(1):83–110, Mar. 2007.
25. J. Niehren, J. Schwinghammer, and G. Smolka. A concurrent lambda calculus with futures. *Theoretical Computer Science*, 364:338–356, 2006.
26. P. C. Ölveczky and J. Meseguer. Semantics and pragmatics of Real-Time Maude. *Higher-Order and Symbolic Computation*, 20(1-2):161–196, 2007.
27. G. A. Papadopoulos and F. Arbab. Coordination models and languages. In *M. Zelkowitz (Ed.), The Engineering of Large Systems*, volume 46 of *Advances in Computers*, pages 329–400. Academic Press, 1998.
28. A. Torjusen, O. Owe, and G. Schneider. Towards integration of XML in the Creol object-oriented language. In F. E. Sandnes, editor, *Proc. Norsk Informatikkonferase 2007 (NIK'07)*, pages 107–111. Tapir Akademisk Forlag, 2007.
29. A. Welc, S. Jagannathan, and A. Hosking. Safe futures for Java. In *Proc. 20th Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'05)*, pages 439–453. ACM Press, 2005.
30. I. C. Yu, E. B. Johnsen, and O. Owe. Type-safe runtime class upgrades in Creol. In R. Gorrieri and H. Wehrheim, editors, *Proc. 8th Intl. Conf. on Formal Methods for Open Object-Based Distributed Systems (FMOODS'06)*, volume 4037 of *Springer*, pages 202–217. Springer, June 2006.