A Type-Safe Model of Adaptive Object Groups *

Joakim Bjørk
University of Oslo, Norway
joakimbj@ifi.uio.no

Dave Clarke
Katholieke Universiteit Leuven, Belgium
dave.clarke@cs.kuleuven.be

Einar Broch Johnsen
University of Oslo, Norway
einarj@ifi.uio.no

Olaf Owe
University of Oslo, Norway
olaf@ifi.uio.no

Services are autonomous, self-describing, technology-neutral software units that can be described, published, discovered, and composed into software applications at runtime. 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. This paper explores service-oriented abstractions such as service adaptation, discovery, and querying in an object-oriented setting. We develop a formal model of adaptive object-oriented groups which offer services to their environment. These groups fit directly into the object-oriented paradigm in the sense that they can be dynamically created, they have an identity, and they can receive method calls. In contrast to objects, groups are not used for structuring code. A group exports its services through interfaces and relies on objects to implement these services. Objects may join or leave different groups. Groups may dynamically export new interfaces, they support service discovery, and they can be queried at runtime for the interfaces they support. We define an operational semantics and a static type system for this model of adaptive object groups, and show that well-typed programs do not cause method-not-understood errors at runtime.

1 Introduction

Good software design often advocates a loose coupling between the classes and objects making up a system. Various mechanisms have been proposed to achieve this, including programming to interfaces, object groups, and service-oriented abstractions such as service discovery. By programming to interfaces, client code can be written independently of the specific classes that implement a service, using interfaces describing the services as types in the program. Object groups loosely organize a collection of objects that are capable of addressing a range of requests, reflecting the structure of real-world groups and social organizations in which membership is dynamic [18]; e.g., subscription groups, work groups, service groups, access groups, location groups, etc. Service discovery allows suitable entities (such as objects) that provide a desired service to be found dynamically, generally based on a query on some kind of interface. An advantage of designing software using these mechanisms is that the software is more readily adaptable. In particular, the structure of the groups can change and new services can be provided to replace old ones. The queries to discover objects are based on interface rather than class, so the software implementing the interface can be dynamically replaced by newer, better versions, offering improved services.

This paper explores service-oriented abstractions such as service adaptation, discovery, and querying in an object-oriented setting. Designing software services and composing services in order to form

*This research was done in the context of the EU project FP7-231620 *HATS*: Highly Adaptable and Trustworthy Software using Formal Models (http://www.hats-project.eu).

To appear in EPTCS.

applications or composite services require abstractions beyond those found in typical object-oriented programming languages. To this end, we develop a formal model of adaptive object-oriented groups that also play the role of service providers for their environment. These groups can be dynamically created, they have identity, and they can respond to methods calls, analogously with objects in the object-oriented paradigm. In contrast to objects, groups are not used for executing code. A group exports its services through interfaces and relies on objects to implement these services. From the perspective of client code, groups may be used as if they were objects by programming to interfaces. However, groups support service-oriented abstractions not supported by objects. In particular, groups may dynamically export new interfaces, they support service discovery, and they can be queried at runtime for the interfaces they support. Groups are loosely assembled from objects: objects may dynamically join or leave different groups. In this paper we develop an operational semantics and a static type system for this adaptive group model based on interfaces, interface queries, groups, and service discovery. The type system ensures that well-typed programs do not cause method-not-understood errors at runtime.

The paper is organized as follows. Section 2 presents the language syntax and a small example. A type and effect system for the language is proposed in Section 3 and an operational semantics in Section 4. Section 5 defines a runtime type system and shows that the execution of well-typed programs is type-safe. Section 6 discusses related work and Section 7 concludes the paper.

2 A Kernel Language for Adaptive Object Groups

We study an integration of service-oriented abstractions in an object-oriented setting by defining a kernel object-oriented language with a Java-like syntax, in the style of Featherweight Java [14]. In contrast to Featherweight Java, types are different from classes in this language: interfaces describe services as sets of method signatures and classes generate objects which implement interfaces. By programming to interfaces, the client need not know how a service is implemented. For this reason, the language has a notion of group which dynamically connects interfaces to implementations. Groups are first-class citizens; they have identities and may be passed around. An object may dynamically join a group and thereby add new services to this group, extending the group's supported interfaces. Objects may be part of several groups. Both objects and groups may join and leave groups, thereby migrating their services between groups. The kernel language considers concurrent objects which interact by synchronous method calls. Concurrent activities are triggered by instantiating classes with `run` methods (similar to overriding the run method of Java's Thread class). This simple concurrency model is relevant for service-oriented systems.

2.1 The Syntax

The syntax of the kernel language is given in Figure 1. A type T in the kernel language is either a basic type, an interface describing a service, or a group of interfaces. A *program* P consists of a list $\overline{IF}$ of interface declarations, a list $\overline{CL}$ of class declarations, and a main block $\{\overline{T}\ \bar{x}; s\}$. The main block introduces a scope with local variables $\bar{x}$ typed by the types $\overline{T}$, and a sequence s of program statements. We conventionally denote by $\bar{x}$ a list or set of the syntactic construct x (in this case, a program variable), and furthermore we write $\overline{T}\ \bar{x}$ for the list of typed variable declarations $T_1\ x_1; \ldots; T_n\ x_n$ where we assume that the length of the two lists $\overline{T}$ and $\bar{x}$ is the same. The types T are the basic type Bool of Boolean expressions, the empty interface Any, the names I of the declared interfaces, and group types $\mathbf{Group}\langle\bar{I}\rangle$ which state that a group supports the set $\bar{I}$ of interfaces. The use of types is further detailed in Section 3, including the subtyping relation and the type system.

Interface declarations IF associate a name I with a set of method signatures. These method signatures may be inherited from other interfaces $\overline{I}$ or they may be declared directly as $\overline{Sg}$. A method *signature* Sg associates a return type T with a name m and method parameters $\overline{x}$ with declared types $\overline{T}$.

Class declarations CL have the form **class** $C(\overline{T}\ \overline{x})$ **implements** $\overline{I}$ $\{\overline{T_1}\ \overline{x_1}; \{\overline{T_2}\ \overline{x_2}; s\}; \overline{M}\}$ and associates a class name C to the services declared in the interfaces $\overline{I}$. In C, these services are realized using methods to manipulate the fields $\overline{x_1}$ of types $\overline{T_1}$. The constructor block $\{\overline{T_2}\ \overline{x_2}; s\}$ initializes the fields, based on the actual values of the formal class parameters $\overline{x}$ of types $\overline{T}$. Remark that the constructor block is executed *asynchronously*. Consequently, it can be used to trigger concurrent activities starting in a new instance of a class. The methods M have a signature Sg and a method body $\{\overline{T}\ \overline{x};\ s; \mathbf{return}\ x;\ \}$ which introduces a *scope* with local variables $\overline{x}$ of types $\overline{T}$ where the sequence of statements s is executed, after which the expression e is returned to the client.

The *expressions e* of the kernel language consist of Java-like expressions for reading program variables x, method calls $x.m(\overline{x})$ where the actual method parameters are given by $\overline{x}$, and object creation **new** $C(\overline{x})$ where the actual constructor parameters are given by $\overline{x}$. Method calls are synchronous and in contrast to Java all method calls are synchronized; i.e., a caller blocks until a method returns and a callee will only accept a remote call when it is idle. For simplicity, the kernel language supports self-calls but not re-entrance (which could be addressed using thread identities as in Featherweight Java [14]). In addition, we consider two expressions which are related to service-oriented software: **newgroup** dynamically creates a new, empty group which does not offer any services to the environment. *Service discovery* may be localized to a named group y: the expression **acquire** I **in** y **except** $\overline{x}$ finds some group g or object o such that g or o offers a service better than I (in the sense of subtyping) and such that g or o is not in the set $\overline{x}$. If the **in** y clause is omitted, then the service provider g or o may be found anywhere in the system.

The *statements s* of the kernel language include standard statements such as **skip**, assignments $x = e$, sequential composition $s_1; s_2$, conditionals, and **while**-loops. To simplify the kernel language, we keep a flat representation of expressions; i.e., expressions must be assigned to program variables before they can be used in other statements. Service interfaces $\overline{I}$ are *dynamically exported* through a group y by the expression x **joins** y **as** $\overline{I}$, which states that object or group x is used to implement the interfaces $\overline{I}$ in the group y. Consequently, y will support the interfaces $\overline{I}$ after x has joined the group. Objects and groups x may try to withdraw service interfaces $\overline{I}$ from a group y by the expression x **leaves** y **as** $\overline{I}$ $\{s_1\}$ **else** $\{s_2\}$. Withdrawing interfaces from a group can lead to runtime exceptions which need to be handled either by the client or by the service provider. In our approach, the exception is handled on the server side; i.e., withdrawing interfaces $\overline{I}$ from y only succeeds if y continues to offer all the interfaces of $\overline{I}$, exported by other objects or groups. Thus, removals may not affect the type of y. If the removal is successful then branch s_1 is taken, otherwise s_2 is taken. In addition, the language includes the statement x **subtypeOf** I y $\{s_1\}$ **else** $\{s_2\}$ which is used to *query* a known group x about its supported interfaces. The statement works like a conditional and branches the execution depending on whether the query succeeds or not. If x offers an interface better than I, the expanded knowledge of the group x becomes available through the variable y in the scope of the statements s_1. If x does not offer an interface as good as I, the branch s_2 is taken. Remark the introduction of a new name for the group inside the scope, which ensures that the knowledge of the extended type is local. (By syntactic sugar, the variable y need not appear in the surface syntax).

2.2 Example

We illustrate the dynamic organization of objects in groups by an example of software which provides text editing support (inspired by [22]). This software provides two interfaces: `SpellChecker` allows

Syntactic Categories.	*Definitions.*		
C : Class name	P	::=	$\overline{IF}\ \overline{CL}\ \{\overline{T}\ \overline{x}; s\}$
I : Interface name	T	::=	Bool | Any | I | **Group**$\langle \overline{I} \rangle$
T : Type name	IF	::=	**interface** I **extends** $\overline{I}\{\overline{Sg}\}$
m : Method name	CL	::=	**class** $C(\overline{T}\ \overline{x})$ **implements** $\overline{I}\ \{\overline{T}\ \overline{x}; \{\overline{T}\ \overline{x}; s\}; \overline{M}\}$
	Sg	::=	$T\ m\ ([\overline{T}\ \overline{x}])$
	M	::=	$Sg\ \{\overline{T}\ \overline{x};\ s; \textbf{return}\ x;\}$
	e	::=	$x \mid x.m(\overline{x}) \mid$ **new** $C(\overline{x}) \mid$ **newgroup** | **acquire** I [**in** x] **except** $\overline{x}$
	s	::=	**skip** $\mid x = e \mid s; s \mid$ **if** $x\ \{s\}$ **else** $\{s\} \mid$ **while** $x\{s\}$
		|	x **joins** x **as** $\overline{I} \mid x$ **leaves** x **as** $\overline{I}\ \{s\}$ **else** $\{s\}$
		|	x **subtypeOf** $I\ x\ \{s\}$ **else** $\{s\}$

Figure 1: Syntax of the kernel language. The type names T include interfaces names I and Bool. Square brackets [] denotes optional elements.

the spell-checking of a piece of text and `Dictionary` provides functionality to update the underlying dictionary with new words, alternate spellings, etc. Apart from an underlying shared catalog of words, these two interfaces need not share state and may be implemented by different classes. Let us assume that the overall system contains several versions of `Dictionary`, some of which may have an integrated `SpellChecker`. Consider a class implementing a text editor factory, which manages groups implementing these two interfaces. The factory has two methods: `makeEditor` dynamically assembles such software into a text editor group and `replaceDictionary` allows the `Dictionary` to be dynamically replaced in such a group. These methods may be defined as follows:

```
Group⟨SpellChecker,Dictionary⟩ makeEditor() {
  Group⟨∅⟩ editor; SpellChecker s; Dictionary d;
  editor = newgroup;
  d = acquire Dictionary except emptyset;
  d subtypeOf SpellChecker ds {
    ds joins editor as Dictionary, SpellChecker;
  } else {
    d joins editor as Dictionary;
    s = new SpellChecker();
    s joins editor as SpellChecker;
  }
  return editor;
}

void replaceDictionary(Group⟨SpellChecker,Dictionary⟩ editor, Dictionary nd){
  Dictionary od;
  nd joins editor as Dictionary;
  od = acquire Dictionary in editor except nd;
  od leaves editor as Dictionary {skip;} else {skip;};
  return;
}
```

The method `makeEditor` acquires a top-level service `d` which exports the interface `Dictionary` (since there is no **`in`**-clause in the **`acquire`**-expression). If `d` also supports the `SpellChecker` interface, we let `d` join the newly created group `editor` as *both* `Dictionary` and `SpellChecker`. Otherwise `d` joins the `editor` group only as `Dictionary`. In this case a new `SpellChecker` object is created and added to the group as `SpellChecker`. Remark that we assumed the presence of several `Dictionary` services in the overall system, otherwise the initial **`acquire`**-expression may not succeed and execution could be

$$\text{(T-VAR)}\quad \Gamma \vdash x : \Gamma(x)$$

$$\text{(T-CALL)}\quad \frac{\Gamma \vdash x : T' \quad \Gamma \vdash \overline{x} : \overline{T} \quad match(m, \overline{T}, T') \quad retType(T', m) = T}{\Gamma \vdash x.m(\overline{x}) : T}$$

$$\text{(T-NEW)}\quad \frac{\Gamma \vdash \overline{x} : ptypes(C) \quad C \prec I}{\Gamma \vdash \texttt{new}\ C(\overline{x}) : I}$$

$$\text{(T-GROUP)}\quad \Gamma \vdash \texttt{newgroup} : \texttt{Group}\langle\emptyset\rangle$$

$$\text{(T-ACQUIRE)}\quad \frac{\Gamma \vdash y : \texttt{Group}\langle S\rangle}{\Gamma \vdash \texttt{acquire}\ I\ \texttt{in}\ y\ \texttt{except}\ \overline{x} : I}$$

$$\text{(T-SUB)}\quad \frac{T \prec T' \quad \Gamma \vdash e : T}{\Gamma \vdash e : T'}$$

Figure 2: The type system for expressions.

blocked at this point. The kernel language could be extended by a more robust version of **acquire** which uses branching (similar to **subtypeOf**); in fact, inside a group g, robustness may be obtained by first checking for the existence of an interface I in g using **subtypeOf** and then binding to the object or group implementing I in g using **acquire**.

The method `replaceDictionary` will replace the `Dictionary` service in a text editor group. First we add the new `Dictionary` service `nd` to the `editor` group and then we fetch the old service `od` in the group by means of an **acquire**, where the **except**-clause is used to avoid binding to the new service `nd`. Finally the old service `od` is removed as `Dictionary` in the group by a `leave` statement. The example illustrates group management by joining and leaving mechanisms as well as service discovery.

3 A Type and Effects System

The language distinguishes behavior from implementations by using an interface as a type which describes a service. Classes are not types in source programs. A class can implement a number of service interfaces, so its instances can export these services to clients. A program variable typed by an interface can refer to an instance of any class which implements that interface. A group typed by $\texttt{Group}\langle\overline{I}\rangle$ exports the services described by the set $\overline{I}$ of interfaces to clients, so a program variable of type I may refer to the group if $I \in \overline{I}$. We denote by Any the "empty" interface, which extends no interface and declares no method signatures. A service described by an interface may consist of only some of the methods defined in a class which implements the interface, so interfaces lead to a natural notion of hiding for classes. In addition to the source program types used by the programmer, class names are used to type the self-reference **this**; i.e., a class name is used as an interface type which exports *all* the methods defined in the class.

Subtyping. The subtype relation $\prec$ is defined as the transitive closure of the extends-relation on interfaces: if I extends J' and $J' \prec J$ or $J' = J$, then $I \prec J$. It is implicitly assumed that all interfaces extends Any, so we let $I \prec$ Any for all I. A group type $\texttt{Group}\langle S\rangle$ is a subtype of I if there is some $J \in S$ such that $J \prec I$, and $\texttt{Group}\langle S\rangle \prec \texttt{Group}\langle S'\rangle$ if for all $J \in S'$ there is some $I \in S$ such that $I \prec J$. We extend the source language subtype relation by letting a class be a subtype of all its implemented interfaces. The reflexive closure of $\prec$ is denoted $\preceq$.

Typing contexts. A typing context Γ binds variable names to types. If Γ is a typing context, x a variable, and T a type, we denote by $dom(\Gamma)$ the set of names which are bound to types in Γ (the domain of Γ) and by $\Gamma(x)$ the type bound to x in Γ. Define the *update* $\Gamma[x \mapsto T]$ of a typing context Γ by $\Gamma[x \mapsto T](x) = T$ and $\Gamma[x \mapsto T](y) = \Gamma(y)$ if $y \neq x$. By extension, if $\overline{x}$ and $\overline{T}$ denote lists $x_1, \ldots, x_n$ and

$$\text{(T-Skip)}\quad \Gamma \vdash \mathtt{skip} : \mathtt{ok}$$

$$\text{(T-Assign)}\quad \frac{\Gamma \vdash e : \Gamma(x)}{\Gamma \vdash x = e : \mathtt{ok}}$$

$$\text{(T-Return)}\quad \frac{\Gamma \vdash s : \mathtt{ok}\langle\Delta\rangle \quad \Gamma \circ \Delta \vdash x : T}{\Gamma \vdash s; \mathtt{return}\ x : T}$$

$$\text{(T-Composition)}\quad \frac{\Gamma \vdash s : \mathtt{ok}\langle\Delta_1\rangle \quad \Gamma \circ \Delta_1 \vdash s' : \mathtt{ok}\langle\Delta_2\rangle}{\Gamma \vdash s; s' : \mathtt{ok}\langle\Delta_1 \circ \Delta_2\rangle}$$

$$\text{(T-Conditional)}\quad \frac{\Gamma(x) = \mathsf{Bool} \quad \Gamma \vdash s_1 : \mathtt{ok}\langle\Delta_1\rangle \quad \Gamma \vdash s_2 : \mathtt{ok}\langle\Delta_2\rangle}{\Gamma \vdash \mathtt{if}\ x\{s_1\}\mathtt{else}\{s_2\} : \mathtt{ok}\langle\Delta_1 \cap \Delta_2\rangle}$$

$$\text{(T-While)}\quad \frac{\Gamma(x) = \mathsf{Bool} \quad \Gamma \vdash s : \mathtt{ok}\langle\Delta\rangle}{\Gamma \vdash \mathtt{while}\ x\{s\} : \mathtt{ok}\langle\Delta\rangle}$$

$$\text{(T-Join)}\quad \frac{\mathit{local}(y) \quad \Gamma(y) = \mathtt{Group}\langle S\rangle \quad \Gamma(x) \prec \bar{I}}{\Gamma \vdash x\ \mathtt{joins}\ y\ \mathtt{as}\ \bar{I} : \mathtt{ok}\langle y \mapsto \mathtt{Group}\langle S \cup \bar{I}\rangle\rangle}$$

$$\text{(T-Leave)}\quad \frac{\begin{array}{c}\Gamma(x) \prec \bar{I} \quad \Gamma(y) = \mathtt{Group}\langle S\rangle \\ \Gamma \vdash s_1 : \mathtt{ok}\langle\Delta_1\rangle \quad \Gamma \vdash s_2 : \mathtt{ok}\langle\Delta_2\rangle\end{array}}{\Gamma \vdash x\ \mathtt{leaves}\ y\ \mathtt{as}\ \bar{I}\ \{s_1\}\ \mathtt{else}\ \{s_2\} : \mathtt{ok}\langle\Delta_1 \cap \Delta_2\rangle}$$

$$\text{(T-Inspect)}\quad \frac{\begin{array}{c}\Gamma(x) = \mathtt{Group}\langle S\rangle \quad y \notin \mathit{dom}(\Gamma) \\ \Gamma[y \mapsto \mathtt{Group}\langle S \cup \{I\}\rangle] \vdash s_1 : \mathtt{ok}\langle\Delta_1\rangle \quad \Gamma \vdash s_2 : \mathtt{ok}\langle\Delta_2\rangle\end{array}}{\Gamma \vdash x\ \mathtt{subtypeOf}\ I\ y\ \{s_1\}\ \mathtt{else}\ \{s_2\} : \mathtt{ok}\langle\Delta_1 \cap \Delta_2\rangle}$$

$$\text{(T-Method)}\quad \frac{\begin{array}{c}\Gamma' = \Gamma[\overline{x} \mapsto \overline{T}, \overline{x'} \mapsto \overline{T'}] \\ \Gamma' \vdash s; \mathtt{return}\ e : T''\langle\Delta\rangle\end{array}}{\Gamma \vdash T''\ m\ (\overline{T}\ \overline{x})\{\overline{T'}\ \overline{x'}; s; \mathtt{return}\ x\} : \mathtt{ok}}$$

$$\text{(T-Class)}\quad \frac{\begin{array}{c}\Gamma[\mathrm{this} \mapsto C, \overline{x_2} \mapsto \overline{T_2}] \vdash \overline{M} : \mathtt{ok} \\ C \prec \bar{I} \quad \Gamma[\mathrm{this} \mapsto C, \overline{x_2} \mapsto \overline{T_2}, \overline{x_1} \mapsto \overline{T_1}, \overline{x_3} \mapsto \overline{T_3}] \vdash s : \mathtt{ok}\langle\Delta\rangle\end{array}}{\Gamma \vdash \mathtt{class}\ C(\overline{T_1}\ \overline{x_1})\ \mathtt{implements}\ \bar{I}\ \{\overline{T_2}\ \overline{x_2}; \{\overline{T_3}\ \overline{x_3}; s\}; \overline{M}\} : \mathtt{ok}}$$

$$\text{(T-Program)}\quad \frac{\begin{array}{c}\Gamma[\overline{x} \mapsto \overline{T}] \vdash s : \mathtt{ok}\langle\Delta\rangle \\ \forall CL \in \overline{CL} \cdot \Gamma \vdash CL : \mathtt{ok}\end{array}}{\Gamma \vdash \overline{IF}\ \overline{CL}\ \{\overline{T}\ \overline{x}; s\} : \mathtt{ok}}$$

Figure 3: The type and effect system for statements, methods, classes, and programs.

$T_1, \ldots, T_n$, we may write $\Gamma[\overline{x} \mapsto \overline{T}]$ for the typing context $\Gamma[x_1 \mapsto T_1] \ldots [x_n \mapsto T_n]$ and $\Gamma[\overline{x_1} \mapsto \overline{T_1}, \overline{x_2} \mapsto \overline{T_2}]$ for $\Gamma[\overline{x_1} \mapsto \overline{T_1}][\overline{x_2} \mapsto \overline{T_2}]$. For typing contexts Γ_1 and Γ_2, we define $\Gamma_1 \circ \Gamma_2$ such that $\Gamma_1 \circ \Gamma_2(x) = \Gamma_2(x)$ if $x \in \mathit{dom}(\Gamma_2)$ and $\Gamma_1 \circ \Gamma_2(x) = \Gamma_1(x)$ if $x \notin \mathit{dom}(\Gamma_2)$.

For typing contexts Γ_1 and Γ_2, we define the *intersection* $\Gamma_1 \cap \Gamma_2$ by $\Gamma_1 \cap \Gamma_2(x) = T$ if T is the best type such that $\Gamma_1(x) = T_1$, $\Gamma_2(x) = T_2$, and $T_1 \preceq T$ and $T_2 \preceq T$. In particular, we have $\Gamma_1 \cap \Gamma_2(x) = \mathtt{Group}\langle S_1 \cap S_2\rangle$ if $\Gamma_1(x) = \mathtt{Group}\langle S_1\rangle$ and $\Gamma_2(x) = \mathtt{Group}\langle S_2\rangle$.

The Type and Effect System. Programs in the kernel language are analyzed using a type and effect system (e.g., [2, 19, 24]). The inference rules for expressions are given in Figure 2 and for statements, methods, classes, and programs in Figure 3.

Expressions are typed by the rules in Figure 2. Let Γ be a typing context. A typing judgment $\Gamma \vdash e : T$ states that the expression e has the type T if the variables in e are typed according to Γ. By T-Var, variables must be typed in Γ. Method calls to a method m on a variable x are typed to T if x has the (interface) type T' such that the types $\overline{T}$ of the actual parameters $\overline{x}$ give a match for m in T' with parameter types $\overline{T}$ and the declared return type of m in T' is T. In T-New, **new** C has type I if the types of the actual parameters to the class constructor can be typed to the declared types of the formal parameters of the class, by means of the auxiliary function *ptypes*, and the class implements I, expressed by $C \prec I$. We omit the definitions of the auxiliary functions *match* and *retType* here, these are straightforward lookup functions on the program's interface table which perform the matching and retrieve the return type of a method in a class, respectively. Similarly, *ptypes* retrieves the types of the formal parameters to a class in the program's class table. By T-Group, a new group has the empty group type (with no exported interfaces). By T-Acquire, service discovery has the obvious type, if successful. The premise of the rule is omitted if the statement has no **in**-clause. Rule T-Sub captures subtyping in the type system.

Statements are typed by the rules in Figure 3. Let Γ and Δ be typing contexts. A typing judgment $\Gamma \vdash s : \mathbf{ok}\langle\Delta\rangle$ expresses that the statement s is well-typed if the variables in s are typed according to Γ and that the typing context for further analysis should be modified according to the *effect* Δ. Empty effects are omitted in the presentation of the rules. The typing of statements **skip** and $x = e$ are standard. These judgments have no effects. The statement **return** x has a return type and is typed in the effect of typing the statements of the method body. The use of effects can be seen in rule T-COMPOSITION, where the second statement is type checked in the typing context modified by the effect of analyzing the first statement, and the effects are accumulated in the conclusion of the rule. Rules T-CONDITIONAL and T-WHILE propagate effects from the subexpressions; in the case of T-CONDITIONAL the resulting effect is approximated by taking the intersection of the effects of the branches. By T-JOIN, when an object joins a group y and contributes interfaces $\bar{I}$ to y, the effect is that the type of y is extended with the interfaces $\bar{I}$. Note the requirement $local(y)$, which expresses that y must be a local variable in the scope of the method being analyzed. (We omit the definition, which is again a lookup in the class table of the program). Without this restriction, a field could dynamically extend its type, resulting in an unsound system; e.g., an assignment `f=e` in a statically well-typed method could become unsound if the type of `f` were extended. However extending the type `T` of a local variable which copies the value of `f` to a type `T'` and assigning the result back to a field `f'` is allowed, as `f'` would need to be of the extended type `T'` and `f` would remain of type `T` as required by the other method. (For comparison, the needed restriction to local variables is handled differently in the query statement **subtypeOf**, which introduces a fresh local variable.) Rule T-LEAVE shows that leaving a group has no effect on the typing context, and the effects of the two branches are treated as for the conditional. Rule T-INSPECT shows how the typing context is extended with a new variable y which extends the type of the group x for the scope of the branch s_1. The overall effect is again the intersection of the effects of the two branches.

Programs, classes, and methods are typed in the standard way. Methods do not have effects, which reflects that effects are constrained to local variables inside methods. Likewise, classes and programs do not have effects. (For simplicity, the standard type checking of interface declarations is omitted in the presentation.) The body of a class constructor and the main method of a program may have the same effects as the body of a method.

4 Operational Semantics

The runtime syntax is given in Figure 4. A runtime configuration *cn* is either the empty configuration ε or it consists of objects *obj* and groups *grp*. Groups *grp* have an identity g and contain a set *export* of interfaces I associated with the objects o implementing them. Objects *obj* have an identity o, a state σ, and a stack ρ of processes *proc*. When an object has processes to execute, it executes the process at the top of its stack. The stack grows with self-calls and shrinks at method returns. The empty stack is denoted *idle*. A state σ maps program variables x to their types T and values v. A process *proc* can be *error* or it has a local state σ and a sequence s;**return** x; of statements to be executed. The expression **wait**(o,m) encodes a *lock*, expressing that the object is waiting for the return value of method m in another object o (or on an auxiliary self-call). Values v include object and group names, and Booleans.

The operational semantics is given by rules in the style of SOS [21], reflecting small-step semantics. Each rule describes one step in the execution of an object. Concurrent execution is given by standard SOS context and concurrency rules (not shown here), and we assume associative and commutative matching over configurations (as in rewriting logic [7]). Thus objects execute concurrently, with the following exceptions: The rule for synchronous remote call (CALL1) refers to both the caller and callee objects

Syntactic Categories.	*Definitions.*		
g : Group name	cn	::=	$\varepsilon \mid grp \mid obj \mid cn\ cn$
o : Object name	grp	::=	$g(export)$
	$export$	::=	$\{o : I\} \mid export \cup export$
	obj	::=	$o(\sigma, \rho)$
	ρ	::=	$idle \mid proc \mid proc; \rho$
	$proc$	::=	$m\{\sigma \mid sr\} \mid error$
	σ	::=	$x \mapsto \langle T, v \rangle \mid \sigma \circ \sigma$
	e	::=	**wait**$(o, m) \mid \ldots$
	sr	::=	$s \mid s;$ **return** $x;$
	v	::=	$o \mid g \mid$ true $\mid$ false

Figure 4: The runtime syntax, extending the language syntax for expressions e and statements s.

and therefore the two objects must *synchronize* and the caller will be blocked by the **wait** statement. Furthermore rules involving an object and a group will lock the group in question, thereby disallowing concurrent execution of other objects involving the same group. This is crucial in the JOIN and LEAVE1 rules for **joins** and **leaves**, which may actually modify the group.

We define the lookup of a program variable x in a state σ by $\sigma(x) = \langle T, v \rangle$, with the projections $\sigma^T(x) = T$ and $\sigma^V(x) = v$. Thus, for a state σ, σ^T gives the associated mapping of program variables to their types and σ^V the mapping of program variables to their values. The SKIP rule is standard and states that a skip has no effect. The effect of assignment is divided into two rules, ASSIGN1 for local variables, updating l, and ASSIGN2 for fields, updating a. In the rule NEW-GROUP, a globally unique group identifier is found by $fresh(g)$. Then an empty group with this identifier is added to the configuration. The two rules COND1 and COND2 handle the two cases of the conditional statement.

Method calls are handled by CALL1 for calls to other objects, CALL2 for self calls, and CALL3 for calls to groups. When a call is made to another object in CALL1, the called object must be in an *idle* state. The caller blocks until the generated **wait** statement can be executed. In the **wait** statement, the callee and method name are recorded, which allows the runtime type system to infer the proper type of the return value from method m in the proper class. Let $bind(m, C, \overline{v})$ denote the process resulting from the activation of method m in C, in which l maps the parameters of m to their declared types and values $\overline{v}$, and the local variables to their declared types and default values. The callee gets the process $bind(m, C, (a \circ l)^V(\overline{y}))$, where C is the class of the callee, pushed onto its process stack ρ. With self calls in CALL2, the process stack cannot be idle, but a **wait** statement replaces the call statement and an instance of the called method is pushed to the stack. In CALL3, a call to a group is reduced to a call to a group or an object *inside* the callee which exports an appropriate interface to the group. By appropriate we mean that the called method is supported by the interface (formally, $m \in mtd(I)$). RETURN1 handles returns from remote calls. Here the blocking **wait** statement is replaced by the returned value. Returns from self calls are handled in a similar way by the RETURN2 rule. (Remark that the generalization to concurrent objects with asynchronous calls and futures is straightforward as in [6, 15] whereas the extension to multi-threaded programs would require re-entrant lock as in [14]).

The **new** statement is handled by the NEW-OBJECT rule, where $fresh(o', C)$ asserts that o' is a new name in the global configuration such that $classOf(o') = C$. An object with this name is created. The mapping $atts(C, \overline{v})$ maps the declared fields of class C to their declared types and default values, *this* to C, and the class parameters to declared types and actual values. The process $init(C)$ corresponds to the

init-block of C, which instantiates local variables to their declared types and default values. The process of the new object is the initial process of its class. Note that an init-block is executed independently from the creator, so it may trigger *active behavior*; for instance, the init-block can call a run method.

The rule JOIN extends the knowledge of a group with the new interfaces from the object's perspective and correspondingly extends the *exports* set from the group's perspective. Service discovery is handled by the ACQUIRE rule. The **`acquire`** expression is replaced by a value v, which is an object or group identifier satisfying the **`in`** and **`except`** clauses. If the **`in`** clause is omitted from the expression, then the premise $(a \circ l)^V(y) = g$ is omitted from the rule. Note that this rule will block if no matching object or group exists. This could be solved by either returning **`null`** (by means of a global check) or by adding an **`else`** branch similar to those in QUERY1 and QUERY2. Within the kernel language, the existence of a matching object or group inside a group can be checked using the query mechanisms.

The **`leaves`** statement is handled by the rules LEAVE1 for a successful leave and LEAVE2 for an unsuccessful one. A group or object x may leave a group successfully if the group provides the same interface support without x. To determine this, we use the function $intf(export)$ which returns a set containing the interfaces of all the pairs in $export$, removing redundant information. An entry is redundant if a subtype of the entry is present in the set. The type of the group does not change by a **`leaves`** statement and hence the object does not need to update information about the group. The branches s_1 or s_2 are chosen depending on the success. The rules QUERY1 and QUERY2 handle the branching statement that checks if a group exports a given interface. If the test succeeds then a fresh variable y is introduced and is only visible in s_1. The type of this variable is the union of what the current object already knew about the group and the new information I. If the test fails the s_2 branch is chosen by QUERY2.

The initial state. For a program $P = \overline{IF}\ \overline{CL}\ \{\overline{T}\ \overline{x}; s\}$, we define the initial state to be $o(\varepsilon, main\{\overline{x} \mapsto \langle \overline{T}, default(\overline{T})\rangle | s; \})$ where o is such that $fresh(o, Main)$.

5 Type Safety

This section extends the type system of Section 3 to runtime configurations and shows that the execution of well-typed programs remains well-typed.

5.1 Well-Typed Configurations

The extension of the type system to runtime configurations is given in Figure 6. The typing context Γ stores the types of all constant values (object and group identities) at runtime. By RTT-CONFIG, a configuration is well-typed if all objects and groups are well-typed. By RTT-GROUP, a group is well-typed if all the objects which export interfaces through the group implement these interfaces (checked by RTT-EXPS and RTT-EXP). By RTT-OBJECT, an object is well-typed if its class is its type in Γ and its state and stack are well-typed in the context of the types of the fields. Substitutions (the state of fields and local variables) are checked by RTT-SUBS and RTT-SUB. The stack is well-typed by RTT-STACK if all its processes are well-typed by RTT-PROC; i.e., the state of local variables and the method body sr are well-typed. Observe that due to the query-mechanism of the language, the types of program variables in two processes which stem from activations of the same method, may differ at runtime. For this reason, the typing context used for typing runtime configurations cannot rely on the statically declared types of program variables. This explains why RTT-PROC extends Γ with the *locally stored typing information* l^T to type check l^V and sr. The effects of the static type system are not needed here, as they are reflected by how the operational semantics updates this local type information. For consistency in the presentation,

(SKIP)
$$o(a,m\{l \mid \texttt{skip};sr\};\rho) \to o(a,m\{l \mid sr\};\rho)$$

(ASSIGN1)
$$\frac{x \in dom(l) \quad l^T(x) = T \quad (a \circ l)^V(y) = v}{o(a,m\{l \mid x = y;sr\};\rho) \to o(a,m\{l[x \mapsto \langle T,v\rangle] \mid sr\};\rho)}$$

(ASSIGN2)
$$\frac{x \notin dom(l) \quad a^T(x) = T \quad (a \circ l)^V(y) = v}{o(a,m\{l \mid x = y;sr\};\rho) \to o(a[x \mapsto \langle T,v\rangle],m\{l \mid sr\};\rho)}$$

(NEW-GROUP)
$$\frac{fresh(g)}{o(a,m\{l \mid x = \texttt{newgroup};sr\};\rho) \to o(a,m\{l \mid x = g;sr\};\rho)\ g(\emptyset)}$$

(COND1)
$$\frac{(a \circ l)^V(x)}{o(a,m\{l|\texttt{if}\ x\ \{s_1\}\ \texttt{else}\ \{s_2\};sr\};\rho) \to (o(a,m\{l|s_1;sr\};\rho)}$$

(COND2)
$$\frac{\neg(a \circ l)^V(x)}{o(a,m\{l|\texttt{if}\ x\ \{s_1\}\ \texttt{else}\ \{s_2\};sr\};\rho) \to o(a,m\{l|s_2;sr\};\rho)}$$

(WHILE)
$$o(a,m\{l \mid \texttt{while}\ x\ \{s_1\};sr\};\rho) \to o(a,m\{l \mid \texttt{if}\ x\ \{s_1\ ;\ \texttt{while}\ x\ \{s_1\}\}\ \texttt{else}\ \{\texttt{skip}\}\ ;sr\};\rho)$$

(CALL1)
$$\frac{(a \circ l)^V(y) = o' \quad classOf(o') = C \quad pr = bind(m,C,(a \circ l)^V(\bar{y}))}{o(a,m\{l \mid x = y.m(\bar{y});sr\};\rho)\ o'(a',idle) \to o(a,m\{l \mid x = \texttt{wait}(o',m);sr\};\rho)\ o'(a',pr)}$$

(CALL2)
$$\frac{(a \circ l)^V(y) = o \quad classOf(o) = C \quad pr = bind(m,C,(a \circ l)^V(\bar{y}))}{o(a,m\{l \mid x = y.m(\bar{y});sr\};\rho) \to o(a,pr;m\{l \mid x = \texttt{wait}(o,m);sr\};\rho)}$$

(CALL3)
$$\frac{(a \circ l)^V(y) = g \quad v : I \in exports \quad m \in mtd(I)}{o(a,m\{l \mid x = y.m(\bar{y});sr\};\rho)\ g(exports) \to o(a,m\{l \mid x = v.m(\bar{y});sr\};\rho)\ g(exports)}$$

(RETURN1)
$$\frac{(a \circ l)^V(x) = v \quad \rho = idle}{o(a,m\{l \mid \texttt{return}\ x;\};\rho)\ o'(a',m'\{l' \mid y = \texttt{wait}(o,m);sr\};\rho') \to o(a,\rho)\ o'(a',m'\{l' \mid y = v;sr\};\rho')}$$

(RETURN2)
$$\frac{(a \circ l)^V(x) = v}{o(a,m\{l \mid \texttt{return}\ x;\};\ m'\{l' \mid y = \texttt{wait}(o,m);sr\};\rho) \to o(a,m'\{l' \mid y = v;sr\};\rho)}$$

(NEW-OBJECT)
$$\frac{fresh(o',C) \quad pr = init(C) \quad a' = atts(C,(a \circ l)^V(\bar{x}))}{o(a,m\{l|x = \texttt{new}\ C(\bar{x});sr\};\rho) \to o(a,m\{l|x = o';sr\};\rho)\ o'(a',pr)}$$

(JOIN)
$$\frac{(a \circ l)^V(x) = v \quad l(y) = \langle \texttt{Group}\langle S\rangle,g\rangle \quad T = \texttt{Group}\langle S \cup \bar{I}\rangle \quad exports' = \bigcup_{I \in \bar{I}}\{v : I\} \cup exports}{o(a,m\{l|x\ \texttt{joins}\ y\ \texttt{as}\ \bar{I};sr\};\rho)\ g(exports) \to o(a,m\{l[y \mapsto \langle T,g\rangle]|sr\};\rho)\ g(exports')}$$

(ACQUIRE)
$$\frac{(a \circ l)^V(y) = g \quad (v : J) \in exports \quad J \prec I \quad v \notin (a \circ l)^V(\bar{x})}{o(a,m\{l \mid x = \texttt{acquire}\ I\ \texttt{in}\ y\ \texttt{except}\ \bar{x};sr\};\rho)\ g(exports) \to o(a,m\{l \mid x = v;sr\};\rho)\ g(exports)}$$

(LEAVE1)
$$\frac{(a \circ l)^V(y) = g \quad (a \circ l)^V(x) = v \quad exports' = exports \setminus \bigcup_{I \in \bar{I}}\{v : I\} \quad intf(exports) = intf(exports')}{o(a,m\{l|x\ \texttt{leaves}\ y\ \texttt{as}\ \bar{I}\ \{s_1\}\ \texttt{else}\ \{s_2\};sr\};\rho)\ g(exports) \to o(a,m\{l|s_1;sr\};\rho)\ g(exports')}$$

(LEAVE2)
$$\frac{(a \circ l)^V(y) = g \quad (a \circ l)^V(x) = v \quad exports' = exports \setminus \bigcup_{I \in \bar{I}}\{v : I\} \quad intf(exports) \neq intf(exports')}{o(a,m\{l|x\ \texttt{leaves}\ y\ \texttt{as}\ \bar{I}\ \{s_1\}\ \texttt{else}\ \{s_2\};sr\};\rho)\ g(exports) \to o(a,m\{l|s_2;sr\};\rho)\ g(exports)}$$

(QUERY1)
$$\frac{y \notin dom(a \circ l) \quad a \circ l(x) = \langle \texttt{Group}\langle S\rangle,g\rangle \quad o' : J \in exports \quad J \prec I}{o(a,m\{l|x\ \texttt{subtypeOf}\ I\ y\ \{s_1\}\ \texttt{else}\ \{s_2\};sr\};\rho)\ g(exports) \to o(a,m\{l[y \mapsto \langle \texttt{Group}\langle S \cup \{I\},g\rangle\rangle]|s_1;sr\};\rho)\ g(exports)}$$

(QUERY2)
$$\frac{(a \circ l)^V(x) = g \quad \texttt{Group}\langle intf(exports)\rangle \not\prec I}{o(a,m\{l|x\ \texttt{subtypeOf}\ I\ y\ \{s_1\}\ \texttt{else}\ \{s_2\};sr\};\rho)\ g(exports) \to o(a,m\{l|s_2;sr\};\rho)\ g(exports)}$$

Figure 5: The operational semantics.

$$\text{(RTT-EMPTY)}\ \Gamma \vdash \varepsilon : \texttt{ok} \qquad \text{(RTT-IDLE)}\ \Gamma \vdash \texttt{idle} : \texttt{ok} \qquad \text{(RTT-WAIT)}\ \Gamma \vdash \texttt{wait}(o,m) : \mathit{retType}(\mathit{classOf}(o),m) \qquad \text{(RTT-DEF)}\ \Gamma \vdash \mathit{default}(T) : T$$

$$\text{(RTT-CONFIG)}\ \frac{\Gamma \vdash cn : \texttt{ok} \quad \Gamma \vdash cn' : \texttt{ok}}{\Gamma \vdash cn\ cn' : \texttt{ok}} \qquad \text{(RTT-GROUP)}\ \frac{\Gamma \vdash \mathit{exports} : \Gamma(g)}{\Gamma \vdash g(\mathit{exports}) : \texttt{ok}} \qquad \text{(RTT-EXP)}\ \frac{I \in S \quad \Gamma(o) \prec I}{\Gamma \vdash o : I : \texttt{Group}\langle S\rangle} \qquad \text{(RTT-SUB)}\ \frac{\Gamma \vdash v : \Gamma(x)}{\Gamma \vdash x \mapsto v : \texttt{ok}} \qquad \text{(RTT-SUBS)}\ \frac{\Gamma \vdash a : \texttt{ok} \quad \Gamma \vdash a' : \texttt{ok}}{\Gamma \vdash a \circ a' : \texttt{ok}}$$

$$\text{(RTT-OBJECT)}\ \frac{\Gamma' = \Gamma \circ a^T \quad \Gamma' \vdash a^V : \texttt{ok} \quad \mathit{classOf}(o) = \Gamma(o) \quad \Gamma' \vdash \rho : \texttt{ok}}{\Gamma \vdash o(a,\rho) : \texttt{ok}} \qquad \text{(RTT-EXPS)}\ \frac{\Gamma \vdash \mathit{exports} : \texttt{Group}\langle S\rangle \quad \Gamma \vdash \mathit{exports}' : \texttt{Group}\langle S\rangle}{\Gamma \vdash \mathit{exports} \cup \mathit{exports}' : \texttt{Group}\langle S\rangle}$$

$$\text{(RTT-PROC)}\ \frac{\Gamma' = \Gamma \circ l^T \quad \Gamma(\mathit{this}) = C \quad \Gamma' \vdash l^V : \texttt{ok} \quad \Gamma' \vdash sr : \mathit{retType}(C,m)}{\Gamma \vdash m\{l|sr;\} : \texttt{ok}} \qquad \text{(RTT-STACK)}\ \frac{\Gamma \vdash \mathit{proc} : \texttt{ok} \quad \Gamma \vdash \rho : \texttt{ok}}{\Gamma \vdash \mathit{proc};\rho : \texttt{ok}}$$

Figure 6: The runtime type system.

the typing of fields is represented in the same way, although these types are not altered by the execution. The rules from the static type checking are reused as appropriate.

5.2 Subject Reduction

The type system guarantees that the type of *fields* in an object never changes at runtime (in particular, recall the restriction $\mathit{local}(y)$ in rule T-JOIN). This allows us to establish in Lemma 1 from the static typing of methods in well-typed programs that method binding, if successful, results in a well-typed process at runtime. To show that the **error** process cannot occur in the execution of well-typed programs, it suffices to show that substitutions are always well-typed. Lemma 2 shows that this is the case for the initial configuration and Lemma 3 shows that one execution step preserves runtime well-typedness. Together, these lemmas establish a subject reduction theorem for the language, expressing that well-typedness is preserved during the execution of well-typed programs and in particular that method binding always succeeds. Here, $\xrightarrow{*}$ denotes the reflexive and transitive closure of the reduction relation $\rightarrow$.

Lemma 1 *Assume that a well-typed program has a class C which defines a method m with formal parameters $\overline{x}$ of type $\overline{T}$ and return type T. Let o be an object such that $\mathit{classOf}(o) = C$ and $\Gamma \vdash o(a,\rho) : \texttt{ok}$. If $\Gamma \vdash \overline{v} : \overline{T}$, then $\Gamma \circ a^T \vdash \mathit{bind}(m,C,\overline{v}) : T$.*

Lemma 2 *Let P be a program such that $\Gamma \vdash P : \texttt{ok}$ and let cn be the initial state of P. Then $\Gamma \vdash cn : \texttt{ok}$.*

Lemma 3 *If $\Gamma \vdash cn : \texttt{ok}$ and $cn \rightarrow cn'$ then there is a Γ' such that $\Gamma' \vdash cn' : \texttt{ok}$ and $\Gamma \subseteq \Gamma'$.*

Theorem 1 (Subject reduction) *Let $\Gamma \vdash P$ and let cn be the initial runtime state of P. If $cn \xrightarrow{*} cn'$ then there is a Γ' such that $\Gamma' \vdash cn' : \texttt{ok}$ and $\Gamma \subseteq \Gamma'$.*

6 Related Work

Object orientation is well-suited for designing small units which encapsulate state with behavior, but does not directly address the organization of more complex software units with rich interfaces. Two approaches to building flexible and adaptive complex software systems involve, independently, object groups and service discovery. Our work unifies these two approaches in a formal, type-safe setting.

The most common use of object groups is to provide replicated services in order to offer better fault tolerance. Communication to elements of a group is via multicast. This idea originated in the Amoeba

operating system [16]. The component model Jgroup/ARM [20] adopts this idea to provide autonomous replication management using distributed object groups. In this setting, members of a group maintain a replicated state for reasons of consistency. The ProActive active object programming model [3] supports abstractions for object groups, which enable group communication—via method call—and various means for synchronizing on the results of such method calls, such as wait-for-one and wait-for-all. ProActive is formalized in Caromel and Henrio's Theory of Distributed Objects [4]. These notions of group differ from ours in two respects. Firstly, in these approaches communication with groups is via multicast, whereas in our approach each message will be delivered to exactly one object, and secondly, in the formal theory, groups are fixed upon creation. Furthermore, there is no notion of service discovery associated with groups.

Object groups have been investigated as a modularization unit for objects which is complementary to components. Groups meet the needs of organizing and describing the statics and dynamics of networks of collaborating objects [18]; groups can have many threads of control, they support roles (or interfaces), and objects may dynamically join and leave groups. Lea [18] presents a number of common usages for groups and discusses their design possibilities, inspired from CORBA. Groups have been used to provide an abstraction akin to a notion of component. For example, in Oracle Siebel 8.2 [8], groups are used as units of deployment, units of monitoring, and units of control when deploying and operating components on Siebel servers. Our approach abstracts from most of these details, though groups are treated as first class entities in our calculus.

Another early work on groups is ActorSpaces [1], which combine Actors with Linda's pattern matching facility, allowing both one-to-one communication, multicast, and querying. Unlike our approach, groups in ActorSpaces are intensional: all actors with the same interface belong to the same group. Furthermore ActorSpaces support broadcast communication to a group, which has not been considered in this paper as it would differentiate communication with an object and with a group. Compared to our paper, these works do not give a formalization of group behavior or discuss typing.

Object groups have further been used for coordination purposes. For example, CoLaS [9] is a coordination model based on groups in which objects may join and leave groups. CoLaS goes beyond the model in our paper by allowing very intrusive coordination of message delivery based on a coordinator state. In our model, the groups don't have any state beyond the state of their objects. Similar to our model, objects enroll to group roles (similar to interfaces). However, unlike our model objects may leave a group at any time, and the coordinator may access the state of participants. The model is implemented in Smalltalk and neither formalization nor typing is discussed [9]. Concurrent object groups have also been proposed to define collaborating objects with a single thread of control in programming and modeling languages [15,23]. Concurrent object groups do not have identity and function as runtime restrictions on concurrency rather than as a linguistic concept.

Microsoft's Component Object Model (COM) supports querying a component to check whether it supports a specific interface, similar to the query-mechanism considered in this paper. A component in COM may also have several interfaces, which are independent of each other. In contrast to the model presented in our paper, COM is not object-oriented and the interfaces of a component are stable (i.e., they do not change). COM has proven difficult to formalize; Pucella develops λ^{COM} [22], a typed λ-calculus which addresses COM components in terms of their interfaces, and discusses extensions to the calculus to capture subtyping, querying for interfaces, and aggregation.

A wide range of service discovery mechanisms exist [13]. The programming language AmbientTalk [10] has built-in service discovery mechanisms, integrated in an object-oriented language with asynchronous method calls and futures. In contrast to our work, AmbientTalk is an untyped language, and lacks any compile time guarantees. Various works formalise the notion of service discovery [17],

but they often do so in a formalism quite far removed from the standard setting in which a program using service discovery would be written, namely, an object-oriented setting. For example, Fiadeiro et al.'s [11] model of service discovery and binding takes an algebraic and graph-theoretic approach, but it lacks the concise operational notion of service discovery formalized in our model. No type system is presented either.

Some systems work has been done that combines groups and service discovery mechanisms, such as group-based service discovery mechanisms in mobile ad-hoc networks [5, 12]. In a sense our approach provides language-based abstractions for a mechanism like this, except that ours also is tied to interface types to ensure type soundness and includes a notion of exclusion to filter matched services.

Our earlier work [6] enabled objects to advertise and retract interfaces to which other objects could bind, using a primitive service discovery mechanism. A group mechanism was also investigated as a way of providing structure to the services. In that work services were equated with single objects, whereas in the present work a group service is a collection of objects exporting their interfaces. In particular, this means that the type of a group can change over time as it comes to support more functionality.

The key differences with most of the discussed works is that the model in this paper remains within the object-oriented approach, multiple groups may implement an advertised service in different ways, and our formalism offers a transparent group-based service discovery mechanism with primitive exclusion policies. Furthermore, our notion of groups has an implicit and dynamically changing interface.

7 Conclusion

The paper has proposed a formal model for adaptive service-oriented systems, based on a notion of object-oriented groups. We develop a kernel object-oriented language in which groups are first-class citizens in the sense that they may play the role of objects; i.e., a reference typed by an interface may refer to an object or to a group. A main advantage is that one may collect several objects into a group, thereby obtaining a rich interface reflecting a complex service, which can be seen as a single object from the outside. Although objects in our language are restricted to executing one method activation at the time, a group may serve many clients at the same time due to inner concurrency.

In contrast to objects, groups may dynamically add support for an increasing number of interfaces. The formation of groups is dynamic; *join* and *leave* primitives in the kernel language allow the migration of services provided by objects and inner groups as well as software upgrade, provided that interfaces are not removed from a group. An object or group may be part of several groups at the same time. This gives a very flexible notion of group.

Adaptive object groups are combined with service discovery by means of *acquire* and *subtypeOf* constructs in the kernel language, which allow a programmer to discover services in an open and unknown environment or in a known group, and to query interface support of a given object or group. These mechanisms are formalized in a general object-oriented setting, based on experiences from a prototype Maude [7] implementation of the group and service discovery primitives. The presented model provides expressive mechanisms for adaptive services in the setting of object-oriented programming with modest conceptual additions. We have developed an operational semantics and type and effects system for the kernel language, and show the soundness of the approach by a proof of type-safety.

The combination of features proposed in this paper suggests that our notion of a group can be made into a powerful programming concept. The work presented in this paper may be further extended in a number of directions. The overall goal of our work is to study an integration of service-oriented and object-oriented paradigms based on a formal foundation. In future work, we plan to extend the

proposed kernel language to multi-thread concurrency and study in more detail how different usages of object groups such as replication, resource, and access groups (see, e.g., [18]) may be captured using the proposed primitives. It is also interesting to study the integration into the kernel language of more service-oriented concepts such as for example error propagation and handling, as well as high-level group management operations such as group aggregation.

References

[1] Gul Agha & Christian J. Callsen (1993): *ActorSpaces: An Open Distributed Programming Paradigm*. In Marina C. Chen & Robert Halstead, editors: *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP)*, ACM, pp. 23–32. Available at http://dx.doi.org/10.1145/155332.155335.

[2] Torben Amtoft, Hanne Riis Nielson & Flemming Nielson (1999): *Type and effect systems - behaviours for concurrency*. Imperial College Press.

[3] Laurent Baduel, Françoise Baude & Denis Caromel (2002): *Efficient, flexible, and typed group communications in Java*. In José E. Moreira, Geoffrey Fox & Vladimir Getov, editors: *Proc. Joint ACM-ISCOPE Conference on Java Grande*, ACM, pp. 28–36. Available at http://doi.acm.org/10.1145/583810.583814.

[4] Denis Caromel & Ludovic Henrio (2005): *A theory of distributed objects - asynchrony, mobility, groups, components*. Springer. Available at http://dx.doi.org/10.1007/b138812.

[5] Dipanjan Chakraborty, Anupam Joshi, Yelena Yesha & Timothy W. Finin (2002): *GSD: a novel group-based service discovery protocol for MANETS*. In: *Proceedings of The Fourth IEEE Conference on Mobile and Wireless Communications Networks*, IEEE, pp. 140–144. Available at http://dx.doi.org/10.1109/MWCN.2002.1045711.

[6] Dave Clarke, Einar Broch Johnsen & Olaf Owe (2010): *Concurrent Objects à la Carte*. In Dennis Dams, Ulrich Hannemann & Martin Steffen, editors: *Concurrency, Compositionality, and Correctness*, *Lecture Notes in Computer Science* 5930, Springer, pp. 185–206. Available at http://dx.doi.org/10.1007/978-3-642-11512-7_12.

[7] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer & Carolyn L. Talcott, editors (2007): *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*. *Lecture Notes in Computer Science* 4350, Springer. Available at http://dx.doi.org/10.1007/978-3-540-71999-1.

[8] Oracle Corporation (2010): *Siebel Business Applications Documentation*. Available at http://www.oracle.com/technetwork/documentation/siebel-087898.html.

[9] Juan Carlos Cruz & Stéphane Ducasse (1999): *A Group Based Approach for Coordinating Active Objects*. In Paolo Ciancarini & Alexander L. Wolf, editors: *Third International Conference on Coordination Languages and Models (COORDINATION'99)*, *Lecture Notes in Computer Science* 1594, Springer, pp. 355–370. Available at http://dx.doi.org/10.1007/3-540-48919-3_25.

[10] Jessie Dedecker, Tom Van Cutsem, Stijn Mostinckx, Theo D'Hondt & Wolfgang De Meuter (2006): *Ambient-Oriented Programming in AmbientTalk*. In Dave Thomas, editor: *Proc. 20th European Conference on Object-Oriented Programming, (ECOOP'06)*, *Lecture Notes in Computer Science* 4067, Springer, pp. 230–254. Available at http://dx.doi.org/10.1007/11785477_16.

[11] José Luiz Fiadeiro, Antónia Lopes & Laura Bocchi (2011): *An abstract model of service discovery and binding*. *Formal Asp. Comput.* 23(4), pp. 433–463.

[12] Zhen guo Gao, Ling Wang, Mei Yang & Xiaozong Yang (2006): *CNPGSDP: An efficient group-based service discovery protocol for MANETs*. *Computer Networks* 50(16), pp. 3165–3182. Available at http://dx.doi.org/10.1016/j.comnet.2005.12.004.

[13] Peer Hasselmeyer (2005): *On Service Discovery Process Types*. In Boualem Benatallah, Fabio Casati & Paolo Traverso, editors: *Proceedings of the Third International Conference on Service-Oriented Computing (ICSOC 2005)*, *Lecture Notes in Computer Science* 3826, Springer, pp. 144–156. Available at http://dx.doi.org/10.1007/11596141_12.

[14] Atsushi Igarashi, Benjamin C. Pierce & Philip Wadler (2001): *Featherweight Java: a minimal core calculus for Java and GJ*. *ACM Transactions on Programming Languages and Systems* 23(3), pp. 396–450. Available at http://dx.doi.org/10.1145/503502.503505.

[15] Einar Broch Johnsen, Reiner Hähnle, Jan Schäfer, Rudolf Schlatte & Martin Steffen (2011): *ABS: A Core Language for Abstract Behavioral Specification*. In Bernhard Aichernig, Frank S. de Boer & Marcello M. Bonsangue, editors: *Proc. 9th International Symposium on Formal Methods for Components and Objects (FMCO 2010)*, *Lecture Notes in Computer Science* 6957, Springer, pp. 142–164. Available at http://dx.doi.org/10.1007/978-3-642-25271-6_8.

[16] M. Frans Kaashoek, Andrew S. Tanenbaum & Kees Verstoep (1993): *Group communication in Amoeba and its applications*. *Distributed Systems Engineering* 1(1), pp. 48–. Available at http://dx.doi.org/10.1088/0967-1846/1/1/006.

[17] Alessandro Lapadula, Rosario Pugliese & Francesco Tiezzi (2008): *Service Discovery and Negotiation With COWS*. *Electronic Notes in Theoretical Computer Science* 200(3), pp. 133–154. Available at http://dx.doi.org/10.1016/j.entcs.2008.04.097.

[18] Doug Lea (1993): *Objects in groups*. Available at http://gee.cs.oswego.edu/dl/groups/groups.html.

[19] J. M. Lucassen & D. K. Gifford (1988): *Polymorphic effect systems*. In: *Proceedings of the 15th Symposium on Principles of Programming Languages (POPL'88)*, ACM Press, pp. 47–57. Available at http://dx.doi.org/10.1145/73560.73564.

[20] Hein Meling, Alberto Montresor, Bjarne E. Helvik & Özalp Babaoglu (2008): *Jgroup/ARM: a distributed object group platform with autonomous replication management*. *Softw., Pract. Exper.* 38(9), pp. 885–923. Available at http://dx.doi.org/10.1002/spe.853.

[21] Gordon D. Plotkin (2004): *A structural approach to operational semantics*. *Journal of Logic and Algebraic Programming* 60-61, pp. 17–139. Available at http://dx.doi.org/10.1016/j.jlap.2004.05.001.

[22] Riccardo Pucella (2002): *Towards a formalization for COM part I: the primitive calculus*. In Mamdouh Ibrahim & Satoshi Matsuoka, editors: *Proceedings of the 2002 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'02)*, ACM, pp. 331–342. Available at http://dx.doi.org/10.1145/583854.582449.

[23] Jan Schäfer & Arnd Poetzsch-Heffter (2010): *JCoBox: Generalizing Active Objects to Concurrent Components*. In Theo D'Hondt, editor: *European Conference on Object-Oriented Programming (ECOOP 2010)*, *Lecture Notes in Computer Science* 6183, Springer, pp. 275–299. Available at http://dx.doi.org/10.1007/978-3-642-14107-2_13.

[24] Jean-Pierre Talpin & Pierre Jouvelot (1992): *Polymorphic Type, Region and Effect Inference*. *Journal of Functional Programming* 2(3), pp. 245–271. Available at http://dx.doi.org/10.1017/S0956796800000393.