Science of Computer Programming ••• (••••) •••–•••

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

A formal model of service-oriented dynamic object groups ☆

Einar Broch Johnsen [a,*], Olaf Owe [a], Dave Clarke [b,c], Joakim Bjørk [a]

[a] University of Oslo, Norway
[b] Uppsala University, Sweden
[c] KU Leuven, Belgium

ARTICLE INFO

Article history:
Received 26 February 2013
Received in revised form 21 October 2014
Accepted 13 November 2014
Available online xxxx

Keywords:
Object orientation
Object groups
Service orientation
Multithreading
Concurrency
Types
Semantics
Type safety

ABSTRACT

Services are autonomous, self-describing, technology-neutral software units that can be published, discovered, queried, and composed into software applications at runtime. Designing and composing software services to form applications or composite services, require 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 dynamic 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 dynamic 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 [1]; 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.

☆ Partly funded by the EU projects FP7-610582 ENVISAGE: Engineering Virtualized Services (http://www.envisage-project.eu) and FP7-612985 UPSCALE: From Inherent Concurrency to Massive Parallelism through Type-based Optimizations (http://www.upscale-project.eu).

* Corresponding author.
E-mail addresses: einarj@ifi.uio.no (E.B. Johnsen), olaf@ifi.uio.no (O. Owe), dave.clarke@it.uu.se (D. Clarke), joakimbj@ifi.uio.no (J. Bjørk).

http://dx.doi.org/10.1016/j.scico.2014.11.014
0167-6423/

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 applications or composite services require abstractions beyond those found in typical object-oriented programming languages. To this end, we develop a formal model of dynamic 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 are *self-describing* in the sense that they 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. Mechanisms for one-to-many communication and service replication for robustness are not the main focus of our model, but are to some degree supported. In this paper we develop an operational semantics and a static type system for a kernel language which captures this model of dynamic object groups, 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.

This paper extends a paper which appeared at FOCLASA 2012 [2]. In the extended version of the paper, the formalized kernel language includes a multithreaded concurrency model with reentrant method calls and a release mechanism which makes the language more expressive. These were not part of the language considered in [2]. The extended paper further expands on the use of inner groups for group management and discusses the diversity of object groups in object-oriented systems, and how different usages fit with the proposed kernel language.

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 different notions of groups from the perspective of the proposed kernel language. Section 7 discusses related work and Section 8 concludes the paper. The details of the type preservation proof are given in Appendix A.

2. A kernel language for dynamic 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 [3]. 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 belong to several groups. Both objects and groups may join and leave groups, thereby migrating their services between groups. Groups offer distribution of work between the implementations of the group's services, because a request to the group can be handled by any object in the group. To study the integration of these service-oriented abstractions, we consider a concurrent kernel language. For a seamless integration with standard object-oriented languages, the kernel language supports multithread concurrency (e.g., [4,5]), but without shared access to objects. However, this concurrency aspect is largely orthogonal to the group abstraction, which would work equally well with the actor-like concurrency of active objects (e.g., [6,7]).

2.1. The syntax

The syntax of the kernel language is given in Fig. 1. A type T is either a basic type, an interface describing a service, or a group of interfaces. The types T are the Null type with the value null, the unit type Void with the value void, the basic type Bool of Boolean expressions, the empty interface Any, the names I of the declared interfaces, and group types Group($\bar{I}$) 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.

A *program* P consists of a list $\overline{IF}$ of interface declarations, a list $\overline{CL}$ of class declarations, and a main block $\{\overline{T}\ \bar{z}; s;$ **return** void; $\}$. We assume that classes and interfaces have distinct names. The main block introduces a scope with local variables $\bar{z}$ typed by the types $\overline{T}$, and a sequence s of program statements. We conventionally denote by $\bar{z}$ a list or set of the syntactic construct z (in this case, a local program variable), and furthermore we write $\overline{T}\ \bar{z}$ for the list of typed variable declarations $T_1\ z_1; \ldots; T_n\ z_n$ where we assume that the length of the two lists $\overline{T}$ and $\bar{z}$ is the same.

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

Class declarations CL have the form **class** $C(\overline{T}_1\ \overline{w}_1)$ **implements** $\bar{I}$ $\{\overline{T}_2\ \overline{w}_2;\ B\ \overline{M}\}$ and associates a class name C to the services declared in the interfaces $\bar{I}$. In C, these services are realized using methods to manipulate the fields $\overline{w_2}$ of types $\overline{T_2}$. The constructor block B has the form $\{\overline{T}\ \bar{z}; s;$ **return** void; $\}$ and initializes the fields, based on the actual values of the

Please cite this article in press as: E.B. Johnsen et al., A formal model of service-oriented dynamic object groups, Sci. Comput. Program. (2015), http://dx.doi.org/10.1016/j.scico.2014.11.014