Science of Computer Programming 77 (2012) 823–847

Contents lists available at SciVerse ScienceDirect

Science of Computer Programming

journal homepage: www.elsevier.com/locate/scico

ASP_{fun}: A typed functional active object calculus

Ludovic Henrio [a], Florian Kammüller [b,c,*], Bianca Lutz [c]

[a] CNRS – I3S – Univ. Nice Sophia-Antipolis – INRIA, Sophia-Antipolis, France

[b] Middlesex University, London, UK

[c] Technische Universität Berlin, Germany

ARTICLE INFO

Article history:
Received 10 November 2009
Received in revised form 31 October 2010
Accepted 28 December 2010
Available online 12 January 2011

Keywords:
Theorem proving
Object calculus
Futures
Distribution
Typing
Binders

ABSTRACT

This paper provides a sound foundation for autonomous objects communicating by remote method invocations and futures. As a distributed extension of ς-calculus we define ASP_{fun}, a calculus of functional objects, behaving autonomously and communicating by a request-reply mechanism: requests are method calls handled asynchronously and futures represent awaited results for requests. This results in an object language enabling a concise representation of a set of active objects interacting by asynchronous method invocations. This paper first presents the ASP_{fun} calculus and its semantics. Then, we provide a type system for ASP_{fun} which guarantees the "progress" property. Most importantly, ASP_{fun} has been formalised; its properties have been formalised and proved using the Isabelle theorem prover and we consider this as an important step in the formalization of distributed languages. This work was also an opportunity to study different binder representations and experiment with two of them in the Isabelle/HOL theorem prover.

1. Introduction

This paper presents a functional active object language featuring asynchronous method calls and futures; it has been formalised in the Isabelle/HOL theorem prover. ASP_{fun} (asynchronous sequential processes) is an extension of the ς-calculus [1] where objects are distributed into several activities, and activities are the units of distribution. Communications toward activities are asynchronous (remote) method calls; and futures are identifiers for the result of such asynchronous invocations. A future represents an evaluation-in-progress in a remote activity. Futures can be transmitted between activities as any object: several activities may refer to the same future. The calculus is said to be functional because method update is realised on a copy of the object: there is no side-effect. The paper also studies a type system for active objects. Typing is a well-studied technique [2]; we prove here a classical typing property, progress, in unusual settings, distributed active objects.

We mechanically proved properties about ASP_{fun} and, since the calculus is abstract, our semantics and mechanisation can be a basis for the analysis of related languages. Distributed active objects represent an abstract notion of concurrently computing and communicating activities. Clearly, finding a combination of objects and concurrency is not new as a notion — related notions are summarized in the following paragraph — but providing a fully formalized and mechanized calculus including typing for this combination is. Mechanical proofs, though more difficult to perform, are more reliable because they should contain no errors. This article shows that theorem proving techniques can handle distributed features of programming languages. Our work is an important step toward the mechanisation of calculi for distributed computing. The calculus is a model for distributed frameworks relying on active objects or on actors as explained below.

* Corresponding author at: Middlesex University, London, UK.
E-mail addresses: Ludovic.Henrio@inria.fr (L. Henrio), f.kammueller@mdx.ac.uk, flokam@cs.tu-berlin.de (F. Kammüller), sowilo@cs.tu-berlin.de (B. Lutz).

0167-6423/$ – see front matter © 2011 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2010.12.008

Object and distribution: the active object model

The underlying principle for distribution considered in this paper originates from Actors [3,4]. Our calculus provides a model of computations that are distributed in the same way as the actor or the active object paradigm. In these paradigms, distributed computation relies on absence of sharing between processes allowing them to be placed on different machines. Those models feature asynchronous RMI-like communications. We detail below some characteristic distributed languages adhering to these principles.

Principles of actors are the following. Each actor is an independent functional process, i.e., an object together with its own thread. Actors interact by asynchronous message passing. They receive messages in their inbox and process them asynchronously. Instead of having an internal state, actors can change their behaviour, i.e., their reaction to received messages. Actors are some form of active objects. Our approach is to take distribution and parallelism notions similar to actors but fit them into a calculus of classical objects. This article introduces a formalisation, both on paper and in a theorem prover, of actor paradigms in the context of ς-calculus.

From the original actor paradigm [5,6,4], several languages have been designed. Some languages directly feature actors, distributed active objects (like the ProActive [7] library), or other derived paradigms. The calculus ASP_{fun} provides a simple model for such languages.

The ASP calculus [8,9] provides understanding and proofs of confluence for asynchronous distributed systems; it is a formalisation of the active object model. In ASP, active objects communicate in an actor-like manner. Additionally, ASP uses *future* objects, i.e., objects for which the real value is being calculated. Syntactically, the ASP calculus is an extension of the **imp**ς-calculus [1,10] with two primitives (*Serve* and *Active*) to deal with distributed objects.

An active object is similar to an actor in the sense that it has a request queue (corresponding to the actor's mailbox), it does not share memory with other active objects, and active objects communicate by messages. For active objects, communications take the form of a remote method invocation that will be treated asynchronously. We call *activity* the set consisting of an active object, its request queue, the set of normal (also called *passive*) objects known by the active objects, and the set of results the active object has computed. Each active object has a *single* thread; only this thread is allowed to access the active object and the passive ones.

Proactive [7] is a Java middleware for distributed computing. It is based on the notion of active objects and is considered as an implementation of the ASP calculus. It is particularly designed for large scale distributed computations (clusters, Grids, or cloud computing). Deployment is based on the notion of virtual nodes and deployment descriptors: when an activity is created, it is associated with a virtual node, and a deployment descriptor file associates virtual nodes to real machines. As active objects do not share memory they provide a good abstraction of location. Finally, an active object is uniquely associated to a location and an application thread (even if several active objects can be placed on the same machine in practise). Active objects act as the unit of both concurrency and distribution. In ProActive, the programmer only cares about splitting its computation into independent active objects that will run in parallel; then the localisation aspect is delegated to a different role: the deployer. It is a key feature of the programming language and the middleware to guarantee that the program behaves the same whatever physical locations are chosen to deploy the active objects.

Also, the Creol [11] language features futures with (multi)-active objects; distribution principles in Creol are quite similar to ASP_{fun} except that Creol is an imperative language with a more complex semantics. Johnsen et al. [11] also advocate the active object paradigm as a model of distributed computation: "The Creol model targets distributed objects by a looser coupling of method calls and synchronization." The mechanised formalisation of an active object language is a major contribution of this paper. Such a formalisation will increase the confidence in the properties of this programming model and our understanding of distributed computation.

Contribution

We define in this paper ASP_{fun}, a calculus of functional active objects with futures. It formalises the notion of active objects presented in the previous paragraph. For example, the behaviour of ProActive active objects follows quite faithfully the semantics of ASP_{fun}, and thus properties proved here can be transferred to this context. Compared to imperative ASP, ASP_{fun} investigates the typing of active objects and ensures progress properties in a functional context.

The language, its type system, and all properties have been completely formalised (http://gforge.inria.fr/scm/viewvc.php/ASPfun/?root=tods-isabelle) and proved in Isabelle/HOL [12]. This formalisation is approximatively 14 000 lines, only 10% dealing with the language definition, and the rest dealing with the proof of ASP_{fun} properties. We also believe that the formalisation of a calculus like ASP_{fun} in a theorem prover will be helpful in the future design of distributed languages and can provide a reliable basis for proofs using paradigms such as distributed objects, futures, remote method invocations, actors, or active objects. Our main contributions are:

- A functional active object calculus with futures and its properties. We illustrate the expressiveness of the calculus on a couple of examples.
- A type system for active object languages.
- An investigation on how to provide a type-safe calculus featuring active objects and futures, where typing ensures progress.