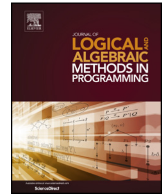
Journal of Logical and Algebraic Methods in Programming 83 (2014) 360–383

Contents lists available at ScienceDirect

Journal of Logical and Algebraic Methods in Programming

www.elsevier.com/locate/jlamp



A sound and complete reasoning system for asynchronous communication with shared futures ☆

Crystal Chang Din, Olaf Owe *

Dept. of Informatics – Univ. of Oslo, P.O. Box 1080 Blindern, N-0316 Oslo, Norway

ARTICLE INFO

Article history:
Received 1 March 2013
Received in revised form 20 February 2014
Accepted 24 March 2014
Available online 5 May 2014

Keywords:
Distributed systems
Compositional reasoning
Hoare Logic
Concurrent objects
Operational semantics
Communication history

ABSTRACT

Distributed and concurrent object-oriented systems are difficult to analyze due to the complexity of their concurrency, communication, and synchronization mechanisms. We consider the setting of concurrent objects communicating by *asynchronous method calls*. The *future mechanism* extends the traditional method call communication model by facilitating sharing of references to futures. By assigning method call result values to futures, third party objects may pick up these values. This may reduce the time spent waiting for replies in a distributed environment. However, futures add a level of complexity to program analysis, as the program semantics becomes more involved.

This paper presents a Hoare style reasoning system for distributed objects based on a general concurrency and communication model focusing on asynchronous method calls and futures. The model facilitates invariant specifications over the locally visible communication history of each object. Compositional reasoning is supported, and each object may be specified and verified independently of its environment. The presented reasoning system is proven sound and (relatively) complete with respect to the given operational semantics.

© 2014 Elsevier Inc. All rights reserved.

1. Introduction

Distributed systems play an essential role in society today. For example, distributed systems form the basis for critical infrastructure in different domains such as finance, medicine, aeronautics, telephony, and Internet services. It is of great importance that such systems work properly. However, quality assurance of distributed systems is non-trivial since they depend on unpredictable factors, such as different processing speeds of independent components. It is highly challenging to test such distributed systems after deployment under different relevant conditions. These challenges motivate frameworks combining precise modeling and analysis with suitable tool support. In particular, *compositional verification systems* allow the different components to be analyzed independently from their surrounding components. Thereby, it is possible to deal with systems consisting of many components.

Object orientation is the leading framework for concurrent and distributed systems, recommended by the RM-ODP [1]. However, method-based communication between concurrent units may cause busy-waiting, as in the case of remote and synchronous method invocation, e.g., Java RMI [2]. Concurrent objects communicating by *asynchronous method calls*, which allows the caller to continue with its own activity without blocking while waiting for the reply, combine object-orientation

☆ This work was done in the context of the EU project FP7-610582 *Envisage: Engineering Virtualized Services* (http://www.envisage-project.eu) and FP7-ICT-2013-X *UpScale: From Inherent Concurrency to Massive Parallelism through Type-based Optimizations* (http://www.upscale-project.eu).

* Corresponding author.
E-mail addresses: crystald@ifi.uio.no (C.C. Din), olaf@ifi.uio.no (O. Owe).

http://dx.doi.org/10.1016/j.jlamp.2014.03.003
2352-2208/© 2014 Elsevier Inc. All rights reserved.

and distribution in a natural manner, and therefore appear as a promising paradigm for distributed systems [3]. Moreover, the notion of *futures* [4–7] improves this setting by providing a decoupling of the process invoking a method and the process reading the returned value. By sharing *future identities*, the caller enables other objects to wait for method results.

ABS is a high-level imperative object-oriented modeling language, based on the concurrency and synchronization model of *Creol* [8]. It supports futures and concurrent objects with an asynchronous communication model suitable for loosely coupled objects in a distributed setting. In *ABS*, each concurrent object encapsulates its own state and processor, and internal interference is avoided as at most one process is executing. The concurrent object model of *ABS* without futures supports compositionality because there is *no direct access* to the internal state variables of other objects, and a method call leads to a new process on the called object. With futures, compositionality is more challenging.

In this paper, we consider the general communication model of *ABS* focusing on the future mechanism. A compositional reasoning system for *ABS* with futures has been presented in [9] based on local communication histories. We here present a revised and simplified version of this system and show that it is sound with respect to an operational semantics which incorporates a notion of global communication history. We also show a completeness result.

The execution of a distributed system can be represented by its *communication history* or *trace*; i.e., the sequence of observable communication events between system components [10,11]. At any point in time the communication history abstractly captures the system state [12,13]. In fact, traces are used in the semantics for full abstraction results (e.g., [14,15]). The *local history* of an object reflects the communication visible to that object, i.e., between the object and its surroundings. A system may be specified by the finite initial segments of its communication histories, and a *history invariant* is a predicate which holds for all finite sequences in the set of possible histories, expressing safety properties [16].

In our reasoning system, we formalize object communication by an operational semantics based on five kinds of communication events, capturing shared (first class) futures, where each event is visible to only one object. Consequently, the local histories of two different objects share no common events. For each object, a history invariant can be derived from the class invariant by hiding the local state of the object. Modularity is achieved since history invariants can be established independently for each object, without interference, and composed at need. This results in behavioral specifications of dynamic system in an open environment. Such specifications allow objects to be specified independently of their internal implementation details, such as the internal state variables. In order to derive a global specification of a system composed of several components, one may compose the specification of different components. Global specifications can then be provided by describing the observable communication history between each component and its environment.

The main contribution of this paper is the presentation of a set of Hoare rules and a proof of soundness and relative completeness with respect to a revised operational semantics including a global communication history. The operational semantics is implemented in Maude by rewriting rules and can be exploited as an executable interpreter for the language, such that execution traces can be automatically generated while simulating programs. In earlier work [17], a similar proof system is derived from a standard sequential language by means of a syntactic encoding. However, soundness with respect to the operational semantics was not considered. A challenge of the current work is that the presence of the global history and shared futures complicate compositional reasoning and also the soundness and completeness proof. We therefore focus the current work on the core communication model with futures and consider process suspension mechanism, but ignore other aspects such as inheritance. The work is relevant for the more general setting of concurrent objects with asynchronous methods and futures, and it can easily be extended to the full *ABS* setting.

An *ABS* reasoning system is currently being implemented within the KeY framework at Technische Universität Darmstadt. The tool support from KeY for (semi-)automatic verification is valuable for verifying *ABS* programs. A publisher–subscriber example will be used here to illustrate the language and the reasoning system.

Paper overview. Section 2 introduces and explains the core language syntax, Section 3 formalizes the observable behavior in the distributed systems, Section 4, presents the operational semantics, and Section 5 defines the proof system for local reasoning within classes and finally considers object composition. A publisher–subscriber example is presented in Section 2 and the corresponding proofs are shown in Section 6. Section 7 defines and proves soundness and relative completeness for our reasoning system. Section 8 shows how to extend the language with non-blocking queries on futures. Section 9 discusses the relevance of choices made in the considered language and formalization, and briefly discusses how some other approaches may affect the reasoning system. Section 10 discusses related and future work, and Section 11 concludes the paper.

2. A core language with shared futures

We consider concurrent objects interacting through method calls. Class instances are concurrent, encapsulating their own state and processor. Each method invoked on the object leads to a new process, and at most one process is executing on an object at a time. Object communication is *asynchronous*, as there is no explicit transfer of control between the caller and the callee. In this setting a *future* represents a placeholder for the return value of a method call. Each future has a unique identity which is *generated* when the method is invoked, and a futures may be seen as a shared entity of information accessible by any object that knows its identity. A future is *resolved* upon method termination, by placing the return value of the method in the future. Thus, unlike the traditional method call mechanism, the callee does not send the return value directly back to the caller. However, the caller may keep a *reference* to the future, allowing the caller to *fetch* the future value

Download English Version:

https://daneshyari.com/en/article/432612

Download Persian Version:

https://daneshyari.com/article/432612

Daneshyari.com