Journal of Logical and Algebraic Methods in Programming 89 (2017) 1–40

Contents lists available at ScienceDirect

Journal of Logical and Algebraic Methods in Programming

www.elsevier.com/locate/jlamp

Behavioural semantics for asynchronous components

R. Ameur-Boulifa [a], L. Henrio [b,*], O. Kulankhina [c], E. Madelaine [c,*], A. Savu [c]

[a] LTCI, Télécom ParisTech, Univ. Paris-Saclay, 75013, Paris, France
[b] Université Côte d'Azur, CNRS, I3S, France
[c] Université Côte d'Azur, Inria, CNRS, I3S, France

ARTICLE INFO

Article history:
Received 2 March 2016
Received in revised form 2 February 2017
Accepted 13 February 2017
Available online xxxx

Keywords:
Behavioural specification
Software components
Distributed systems
Futures

ABSTRACT

Software components are a valuable programming abstraction that enables a compositional design of complex applications. In distributed systems, components can also be used to provide an abstraction of locations: each component is a unit of deployment that can be placed on a different machine. In this article, we consider this kind of distributed components that are additionally loosely coupled and communicate by asynchronous invocations.

Components also provide a convenient abstraction for verifying the correct behaviour of systems: they provide structuring entities easing the correctness verification. This article provides a formal background for the generation of behavioural semantics for asynchronous components. It expresses the semantics of hierarchical distributed components communicating asynchronously by requests, futures, and replies; this semantics is provided using the pNet intermediate language. This article both demonstrates the expressiveness of the pNet model and formally specifies the complete process of the generation of a behavioural model for a distributed component system. The purpose of our behavioural semantics is to allow for verification both by finite instantiation and model-checking, and by techniques for infinite systems.

© 2017 Elsevier Inc. All rights reserved.

1. Introduction

Ensuring the safety of distributed applications is a challenging task. Both the network and the underlying infrastructure are not reliable, and additionally, even without failures, applications are complicated to design because of the multiple execution paths possible. To ensure the safety of distributed applications, we propose to use formal methods to be able to verify the correct behaviour of distributed applications. Consequently, it is necessary to choose a programming abstraction that is convenient to write applications, but also that provides enough information to be able to check the properties of the program. We adopt a programming model that is expressive enough to program complex distributed applications but with some constraints enabling the behavioural verification of these applications. This programming model relies on the notion of distributed software components. Component models provide a structured programming paradigm, and ensure a very good re-usability of programs. Indeed in component applications, both required and provided functionalities are defined by means of provided/required ports; this improves the program specification and thus its re-usability. Several effective distributed component models have been specified, developed, and implemented in the last years [1–4] ensuring

* Corresponding authors.
E-mail addresses: rabea.ameur-boulifa@telecom-paristech.fr (R. Ameur-Boulifa), ludovic.henrio@cnrs.fr (L. Henrio), oleksandra.kulankhina@inria.fr (O. Kulankhina), eric.madelaine@inria.fr (E. Madelaine).

http://dx.doi.org/10.1016/j.jlamp.2017.02.003
2352-2208/© 2017 Elsevier Inc. All rights reserved.

different kinds of properties to their users. Component models have been chosen as the target programming model for many developments in formal methods. Indeed, additionally to the valuable software engineering methodology they ensure, components also provide structural information that facilitate the use of formal methods: the architecture of the application is defined statically.

Distributed component models. Even if our theorems and results can be adapted to most component models, we primarily focus on one distributed component model, the GCM (Grid Component Model [4]). This component model originates from the Grid computing community, it is particularly targeted at composing large-scale distributed applications. Its reference implementation GCM/ProActive relies on the notion of active objects, and ensures that, during execution, each thread is isolated in a single component. Because of active objects, components that usually structure the application composition also provide the structure of the application at runtime, in terms of location and thread (a single applicative thread manipulates the state of the component). We call this kind of components *asynchronous components* because they are loosely coupled entities communicating by a request–reply mechanism. All those aspects facilitate the use of formal methods for ensuring safe behaviour of applications, but they also require specific developments to produce a formal model of an application built from such components.

Contribution. This article formalises the construction of a behavioural model for the core constructs of GCM/ProActive components. It describes formally how we can generate a behavioural model in terms of *pNets* (parameterised Networks of synchronised automata) [5] from the description of the architecture of a GCM/ProActive application and the description of the behaviour of each service method implemented by the programmer. pNets are networks of LTSs with parameters organised (and synchronised) in a hierarchical way. In this article we formalise the automatic construction of the behavioural model for communication, management, and composition aspects.

Our behavioural models are *parameterised*: pNets specify a structured composition of labelled transition systems (LTS) that can use parameters/variables. Each pNet is either formed of other pNets or is a single LTS. Parameters can be used as local variables in an LTS; but they can also be used to define families of pNets of variable size, and to specify synchronisation between pNets (see Section 4). Once the parameterised behavioural model is generated, we can for example generate a finite instance of the model that can be checked against correctness formulas using a model-checking tool. But our behavioural model is richer than what can be checked by finite-state model-checkers and other verification techniques can also be used. For example we are currently working on more symbolic techniques mixing bisimulation algorithms with satisfiability engines to deal with properties of pNet systems with unbounded data, or with unknown sub-nets [6].

The GCM model and its GCM/ProActive implementation provide a very rich environment for building distributed applications, including too many features to be fully formalised in this article. In section 2, we shall define formally a "*Core GCM*" model, containing its most important constructs, namely:

- Primitive components: at the leaves of the hierarchy, from the definition of the service methods, we specify a component able to receive requests and serve each of them one after the other. When a request service terminates, a reply is sent back to the component that emitted the request. The crucial parts composing the model of a primitive component are: the request queue, the handling of asynchronous communications for sending requests and replies, the futures and their management (Section 4.1).
- Composite components (composites, for short): as our component model is hierarchical, a component can be built from the composition of other components. In GCM, composites are instantiated at runtime, it is thus necessary to specify their behaviour in our model too (see Section 4.2). Each composite is in fact implemented as an active object and thus the internal structure of a composite is very similar to the one of a primitive component.
- Component composition: from an ADL (architecture description language) specification, we generate the synchronisations corresponding to the communications that can occur between the different components.
- Futures: futures are frequently used in active object languages, they are place-holders for results of asynchronous invocations, called requests here. We encode in our models the mechanisms for dealing with futures (Section 4.1.5) and the transmission of futures references between components (*first class futures*, Section 5).

Previous works. This article is built upon previous works of the authors. The pNets have already been defined formally in [5], but in a more complex version using a specific form of controllers named "transducers". In this article, in Section 3, we shall provide a new definition, simpler and more concise.

The modelling of basic component features has been addressed in previous publications, including: 1) the pNets-based semantics of primitive and composite components, and their hierarchical composition have been described in [7] and [5]; 2) behavioural models for first-class futures have been studied in [8].

In these works, we built pNet models for specific features of GCM in the context of specific case-studies and proved properties of the studied applications. To illustrate the kind of properties we are able to verify, we proved by model-checking that a master–slave fault-tolerant application [9] behaves correctly: 1) it answers to requests: we proved both reachability and (fair) inevitability of termination of services, 2) the answers (values returned by services) are correct. This shows that our approach addresses both safety and liveness properties, but also functional correctness, modulo data abstraction, based on user requirements.

In Sections 4 and 5 we provide a general formalisation of the features previously studied in those examples.

Download English Version:

https://daneshyari.com/en/article/4951420

Download Persian Version:

https://daneshyari.com/article/4951420

Daneshyari.com