Journal of Logical and Algebraic Methods in Programming 84 (2015) 505–533

Contents lists available at ScienceDirect

Journal of Logical and Algebraic Methods in Programming

www.elsevier.com/locate/jlamp

Executable rewriting logic semantics of Orc and formal analysis of Orc programs

Musab A. AlTurki [a,*,1], José Meseguer [b,2]

[a] King Fahd University of Petroleum and Minerals, Dhahran 31261, Saudi Arabia
[b] The University of Illinois at Urbana-Champaign, Urbana, IL 61801, USA

ARTICLE INFO

Article history:
Received 8 November 2013
Received in revised form 29 January 2015
Accepted 26 March 2015
Available online 3 April 2015

Keywords:
Rewriting logic
Orc
Executable semantics
Maude
Formal analysis
Service orchestration

ABSTRACT

The Orc calculus is a simple, yet powerful theory of concurrent computations with great versatility and practical applicability to a very wide range of applications, as it has been amply demonstrated by the Orc language, which extends the Orc calculus with powerful programming constructs that can be desugared into the underlying formal calculus. This means that for: (i) theoretical, (ii) program verification, and (iii) language implementation reasons, the *formal semantics* of Orc is of great importance. Furthermore, having a semantics of Orc that is *executable* is essential to provide: (i) a formally-defined *interpreter* against which language implementations can be validated, and (ii) a (semi-)automatic way of generating a wide range of *semantics-based* program verification tools, including model checkers and theorem provers.

This work proposes a *formal executable semantics* for Orc in rewriting logic, to support formal verification of Orc programs and to make possible semantics-based correct-by-construction Orc implementations. While being a very simple calculus, Orc has a quite *subtle* semantics, so that fully capturing all its semantic aspects is highly nontrivial. The two main sources of subtlety are: (i) its real-time semantics, and (ii) the priority of *internal* computations within an Orc expression over *external* computations that process responses from *external sites*. In this paper, we show a simple and elegant way of handling these two sources of subtlety in rewriting logic using an order-sorted type system supporting subtypes and subtype polymorphism, and "tick" rewrite rules for capturing time. Moreover, our rewriting semantics incorporates useful *semantic equivalences* between Orc programs as equations and equational attributes, making the semantics both more abstract and more efficient. The semantics of Orc is given in two different styles: (i) an *SOS style*, which is directly based on the original SOS of Orc, whose correctness follows immediately by construction, and (ii) a *reduction semantics*, which is much more efficiently executable and analyzable, as shown through several experiments, and whose correctness is proved using a strong bisimulation theorem. The paper also presents MOrc, a simulator and model checking tool based on the rewriting semantics of Orc and Real-Time Maude. MOrc facilitates formal verification of Orc programs, and allows for user-defined state predicates and LTL formulas, with no need for any prior knowledge of Maude or its rewriting logic foundations.

* Corresponding author.
E-mail addresses: musab@kfupm.edu.sa (M.A. AlTurki), meseguer@cs.illinois.edu (J. Meseguer).

[1] Supported by KFUPM Grant JF121005.
[2] Supported by AFOSR Contract FA8750-11-2-0084 and NSF Grants CCF 09-05584 and CNS 13-19109.

http://dx.doi.org/10.1016/j.jlamp.2015.03.003

1. Introduction

The Orc concurrency calculus [59,61,82] is very remarkable in that it combines great simplicity and mathematical elegance with great versatility and practical applicability to a very wide range of concurrent programming [61,42], web-based programming [61], business processes [22], and distributed cyber-physical system applications. Indeed, the great elegance and naturalness with which applications in all these areas can be programmed has been amply demonstrated by the Orc language [41,60], which extends the Orc calculus with powerful programming constructs. The Orc language's relationship to the Orc calculus can be viewed as the analogue of the relationship between a purely functional language and the lambda calculus: in both cases, all the language's constructs can be desugared into the underlying formal calculus, which is crucial for both ease of reasoning and ease of developing correct implementations.

All this means that for: (i) theoretical, (ii) program verification, and (iii) language implementation reasons, the *formal semantics* of Orc is of great importance. Furthermore, there is by now overwhelming evidence in various approaches to formal semantics that the by far most useful semantic definitions are *executable formal semantic definitions*, for two main reasons. First, many complex languages do not have a formal semantics at all, but have at best lengthy and ambiguous standards in natural language and various compilers, often exhibiting different behaviors. In such a case, a "paper semantics" is of very limited use, since the language's complexity makes it in practice virtually impossible to validate whether such a semantics (which cannot be compared to any other one by the language's lack of a formal semantics) really captures the intended, informal semantics described in its standard. By contrast, an executable formal semantics automatically provides a formally-defined *interpreter*, so that the semantic definition can be validated against the language's standard and against other mature language implementations to ensure that the informal semantics has been correctly formalized. Indeed, this is exactly the approach advocated in the rewriting logic semantics project [51,56,73,57], where large languages have been given an executable semantics in rewriting logic, including Java and the JVM [27,28], Scheme [49], Verilog [50], and, more recently, C [26]. The case of C is a good illustration of why an efficient executable semantics is essential, since the work in [26] is the first formal semantics ever given of the entire C language, a semantics which has been validated against the entire gnu C compiler torture suite and has uncovered bugs in several C tools.

A second reason why an executable formal semantics is enormously more useful than a "paper semantics" is that a wide range of program verification tools, including model checkers, [51,56,57], theorem provers [57,70] and static analysis tools [56,57] can be based *directly* on such an executable formal semantics in the sense of both being generated from the executable semantics and embodying such a semantics. This, in turn, has two main advantages: (i) using languages such as Maude [21] many of these program analysis tools can be generated automatically and for free from the semantic definition: for example, the JavaFAN model checkers for Java and the JVM are automatically generated that way and compare favorably with well-known Java model checkers [27,28], and (ii) there is no gap between the formal semantics and the program analysis tools, since the tools are based on the program semantics. This is by no means the case for most formal tools, which embody such a semantics only implicitly and sometimes erroneously. For example, the work in [26] has uncovered several semantic errors in well-known theorem provers for C, and the work in [50] uncovered similar problems in mature Verilog tools.

The goal of this work is to propose a *formal executable semantics* for Orc in rewriting logic [52], and to exploit such an executable semantics in the above-mentioned ways to support formal verification of Orc programs and to make possible semantics-based correct-by-construction Orc implementations. Compared with a language such as C, the task is in several ways much simpler, since Orc is a much simpler language than C, and has from the very beginning been designed as a formal calculus with an SOS semantics [61]. However, Orc, while being a very simple calculus, has a quite *subtle* semantics, so that fully capturing all its semantic aspects is highly nontrivial. In fact, it is not at all clear that this can be done in just any semantic framework. The sources of subtlety include the following:

1. **Real Time.** Orc is a real-time calculus, where the passage of time is essential and where sophisticated real-time concurrent applications can be developed. Any semantic framework not supporting such a real-time semantics will be useless.
2. **Internal vs. External Computation.** An Orc expression evaluates its constructs internally; but the evaluation may involve making calls to *external sites* and eventually receiving answers from such site calls, so that the internal evaluation can proceed. For example, an Orc expression may invoke both the CNN and BBC web sites with a given timeout, and then send zero, one, or two emails to a given user with the respective contents of the web sites which responded before the specified timeout. To avoid undesirable behaviors, internal computations should always be given priority over external ones. In [61], this is modeled by having *two transition relations* in Orc's SOS semantics, $\hookrightarrow_R$ and $\hookrightarrow_A$.

We show in Section 5 that these two sources of subtlety can be handled in a quite simple and elegant way by our rewriting logic semantics. Specifically, we show that using an order-sorted type structure [32], supporting subtypes and subtype polymorphism, completely solves subtlety (2), so that a single transition relation enforces the desired priority of internal over external computation. The solution of subtlety (1) is even simpler: the addition of a *single* additional "tick" rule, modeling time elapse, to the semantic rules describing Orc's instantaneous computations and the proper definition of time execution semantics and time delays are all that is needed to obtain a real-time semantics.