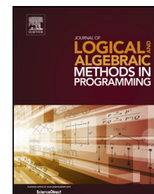
Journal of Logical and Algebraic Methods in Programming ••• (••••) •••–•••

Contents lists available at ScienceDirect

Journal of Logical and Algebraic Methods in Programming

www.elsevier.com/locate/jlamp



Automatic distributed code generation from formal models of asynchronous processes interacting by multiway rendezvous

Hugues Evrard [a,b,*], Frédéric Lang [a,b]

[a] *Inria, Univ. Grenoble Alpes, LIG, F-38000 Grenoble, France*
[b] *CNRS, LIG, F-38000 Grenoble, France*

A R T I C L E   I N F O

Article history:
Received 31 July 2015
Received in revised form 21 February 2016
Accepted 5 September 2016
Available online xxxx

Keywords:
Multiway rendezvous
Compilation
Process algebras
Distributed systems

A B S T R A C T

Formal process languages inheriting the concurrency and communication features of process algebras are convenient formalisms to model distributed applications, especially when they are equipped with formal verification tools (e.g., model checkers) to help hunting for bugs early in the development process. However, even starting from a fully verified formal model, bugs are likely to be introduced while translating (generally by hand) the concurrent model—which relies on high-level and expressive communication primitives—into the distributed implementation—which often relies on low-level communication primitives. In this paper, we present DLC, a compiler that enables distributed code to be generated from models written in a formal process language called LNT, which is equipped with a rich verification toolbox named CADP, and where processes interact by value-passing multiway rendezvous. The generated code uses an elaborate protocol to implement rendezvous, and can be either executed in an autonomous way (i.e., without requiring additional code to be defined by the user), or connected to external software through user-modifiable C functions. The protocol itself is modeled in LNT and verified using CADP. We present several experiments assessing the performance of DLC, including the Raft consensus algorithm.

© 2016 Elsevier Inc. All rights reserved.

1. Introduction

Distributed systems often consist of several concurrent processes, which interact to achieve a global goal. The activity of programming concurrent interacting processes is recognized as complex and error-prone. One way to detect bugs early is to (a) produce a model of the system in a language with well-defined semantics, and to (b) use formal verification methods (e.g., model checking) to hunt for bugs in the model. However, formal models of distributed systems must eventually be translated into a distributed implementation. If this translation is done by hand then semantic discrepancies may appear between the model and the final implementation, possibly leading to bugs. In order to avoid such discrepancies, an automatic translator, i.e., a compiler, can be used.

Such a compiler takes a formal model as input and generates a runnable program, which behaves according to the model semantics. In the case of distributed systems, we want to produce several programs, which can be executed on distinct machines, from a single model of a distributed system. We identified several challenges related to this kind of compilation.

* Corresponding author.
E-mail addresses: Hugues.Evrard@inria.fr (H. Evrard), Frederic.Lang@inria.fr (F. Lang).

http://dx.doi.org/10.1016/j.jlamp.2016.09.002
2352-2208/© 2016 Elsevier Inc. All rights reserved.

Please cite this article in press as: H. Evrard, F. Lang, Automatic distributed code generation from formal models of asynchronous processes interacting by multiway rendezvous, J. Log. Algebraic Methods Program. (2016), http://dx.doi.org/10.1016/j.jlamp.2016.09.002

First, formal models generally rely on concurrency theory operators to express complex interactions between processes, whereas implementation languages often offer only low-level communication primitives. Hence, the complex interactions have to be implemented by non-trivial protocols built upon the low-level primitives, which may be hard to master by (even experienced) programmers. As a brief example, the synchronization of n distributed processes may be expressed by a single rendezvous primitive (high-level), while it requires a protocol between the n processes when only message passing primitives (low-level) are available. For any process interaction specified in the high-level model, the compiler must be able to automatically instantiate such protocols in the generated code.

Second, the generated programs should be able to interact with their environment. Such interactions are often abstracted away in the formal models, while a real interaction is required in the final implementation. For instance, consider a distributed system where some process deals with a database. In the formal model, the database may be abstracted away by read and write operations. However, we want the implementation of these processes to actually connect to an external database which is developed independently from the distributed system under study. The compiler should provide a mechanism to define interactions with the external environment and embed them in the final implementation.

Third, the generated implementation must take benefit of the distributed nature of the system to achieve reasonable performances for rapid prototyping. Performance not only depends on the speed of each process, but also on how process interaction is implemented. Naive implementations can lead to very inefficient executables, due to unforeseen bottlenecks. For instance, a compiler implementing a naive protocol that consists in acquiring a unique global lock to proceed on process interaction would be extremely inefficient as processes would mostly waste time waiting for the lock while they often could safely execute concurrently. An efficient and decentralized protocol is therefore required to enable decent execution times. Even though the aim is not to compete with hand-crafted optimized implementations, a too important performance penalty would make the rapid prototyping approach irrelevant.

In this paper, we consider models written in LNT [12], a process language with formal semantics. LNT combines a user-friendly syntax, close to mainstream imperative languages, together with communication and concurrency features inherited from process algebras, in particular the languages LOTOS [31] and E-LOTOS [32]. Its semantics are formally defined in terms of an LTS (*Labeled Transition System*): the observable events of an LNT process are *actions* (possibly parametrized with data) on *gates* (which represent ports of interaction between processes, and also with the environment), which label the transitions between states of the process.

LNT models can be formally verified using software tools available in the CADP[1] (*Construction and Analysis of Distributed Processes*) [26] tool box, which provides simulation, model checking, and test generation tools, among others.

LNT enables a high-level description of nondeterministic concurrent processes that run asynchronously (i.e., at independent speeds, as opposed to synchronous processes driven by a global clock), and that interact by *value-passing rendezvous* (or *synchronization*) on actions. The value-passing rendezvous mechanism of LNT is expressive and general:

- A rendezvous may involve any number of processes (*multiway* rendezvous), i.e., it is not restricted to binary synchronizations. LNT even features n-among-m synchronization [27], in which a rendezvous may involve any subset of n processes out of a larger set of m.
- Due to nondeterminism (select statement), every process may be ready for several actions at the same time. Different rendezvous may thus involve one or more common processes, in which case we say that the rendezvous are *conflicting*. Therefore, for a rendezvous between processes to occur actually, it is not enough that all processes are ready; they must also all simultaneously agree to take that rendezvous instead of conflicting ones.
- Processes may exchange data during the rendezvous (*value-passing* rendezvous). Each data exchange may involve an arbitrary number of senders and receivers, and a given process may simultaneously send and receive different pieces of data during the same rendezvous.

The research problem we tackle here is how to automatically generate a distributed implementation from an LNT model of a distributed system. To our knowledge, there does not exist an automatic distributed code generation tool for a formal language that not only features such a general rendezvous mechanism, but is also equipped with powerful verification tools. We introduce DLC[2] (*Distributed LNT Compiler*), a new tool that achieves automatic generation of a distributed implementation in C from an LNT model. We focus on LNT since we think its roots in process algebra offer a well-grounded basis for formal study of concurrent systems [22], and because it is already equipped with the numerous verification features of our team's toolbox CADP, which however still lacks distributed rapid prototyping. Nonetheless, our approach should be relevant to any language whose inter-process communication and synchronization primitive is value-passing multiway rendezvous. DLC meets the three challenges stated earlier:

- DLC transforms each concurrent process of the distributed system model into a sequential program, and instantiates an elaborate protocol to handle value-passing multiway rendezvous. We designed a rendezvous protocol that combines

[1] http://cadp.inria.fr.
[2] http://hevrard.org/DLC.

Please cite this article in press as: H. Evrard, F. Lang, Automatic distributed code generation from formal models of asynchronous processes interacting by multiway rendezvous, J. Log. Algebraic Methods Program. (2016), http://dx.doi.org/10.1016/j.jlamp.2016.09.002

Download English Version:

https://daneshyari.com/en/article/4951412

Download Persian Version:

https://daneshyari.com/article/4951412

Daneshyari.com