Science of Computer Programming 115–116 (2016) 199–224

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

Global consensus through local synchronization: A formal basis for partially-distributed coordination

S.-S.T.Q. Jongmans [a,b,*], F. Arbab [b,a]

[a] Universiteit Leiden, Niels Bohrweg 1, 2333 CA, Leiden, Netherlands
[b] Centrum Wiskunde & Informatica, Science Park 123, 1098 XG, Amsterdam, Netherlands

ARTICLE INFO

Article history:
Received 9 March 2014
Received in revised form 7 August 2015
Accepted 14 September 2015
Available online 24 September 2015

Keywords:
Reo
Port automata
Nonassociative product
Parallel composition
Connector implementation

ABSTRACT

A promising new application domain for coordination languages is expressing interaction protocols among threads/processes in multicore programs: coordination languages typically provide high-level constructs and abstractions that more easily compose into correct (with respect to a programmer's intentions) protocol specifications than do low-level synchronization constructs (e.g., locks, semaphores, etc.) provided by conventional languages. However, a crucial step toward adoption of coordination languages for multicore programming is the development of compiler technology: programmers must have tools to automatically generate efficient code for high-level protocol specifications.

In ongoing work, we are developing compilers for a coordination language, Reo, based on that language's automata semantics. As part of the compilation process, our tool computes the product of a number of automata, each of which models a constituent of the protocol to generate code for. This approach ensures that implementations of those automata at runtime reach a consensus about their global behavior in every step. However, this approach has two problems: state space explosion at compile-time and oversequentialization at runtime. In this paper, we provide a solution by defining a new, local product operator on those automata that avoids these problems. We then identify a sufficiently large class of automata for which using our new product instead of the existing one is semantics-preserving.

1. Introduction

Context Coordination languages have emerged for the specification and implementation of interaction protocols among concurrent processes (services, threads, etc.). This class of languages includes Reo [1,2], a graphical language for compositional construction of protocols, manifested as *connectors*: communication media through which processes can interact with each other. Fig. 1 shows example connectors in their usual graphical syntax. Briefly, connectors consist of one or more *channels*, through which data items flow, and a number of *nodes*, on which channel ends coincide. Through connector *composition* (the act of gluing connectors together on their shared nodes), users can construct arbitrarily complex connectors. To communicate data and synchronize their execution, concurrent processes perform I/O-operations on the *boundary nodes* of a connector (shown in Fig. 1 as open circles).

* Corresponding author at: Centrum Wiskunde & Informatica, Science Park 123, 1098 XG, Amsterdam, Netherlands.
E-mail addresses: jongmans@cwi.nl (S.-S.T.Q. Jongmans), farhad@cwi.nl (F. Arbab).

http://dx.doi.org/10.1016/j.scico.2015.09.001
0167-6423/

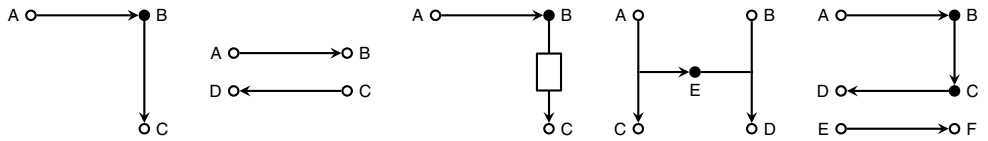

Fig. 1. Five example connectors. Open circles represent boundary nodes, on which processes perform I/O-operations; filled circles represent nodes for internal routing. The first four connectors in this figure consist of two *primitives* (i.e., minimal subconnectors); the fifth connector consists of four primitives. The two connected primitives in the first, third, and fourth connector have one shared node; the three connected primitives in the fifth connector have two shared nodes.

Reo has successfully been used to confront the problem of Web service composition, both theoretically [3,4] and practically [5,6]. Perhaps even more promising and exciting, however, is recent work on using Reo for programming multicore applications. When it comes to multicore programming, Reo has a number of advantages over conventional programming languages, which feature a fixed set of low-level synchronization constructs (locks, mutexes, etc.). Programmers using such a conventional language have to translate the synchronization needs of their protocols into the synchronization constructs of that language. Because this translation occurs in the mind of the programmer, invariably some context information either gets irretrievably lost or becomes implicit and difficult to extract in the resulting code. In contrast, Reo allows programmers to compose the protocols of their application (i.e., connectors) at a high abstraction level. Not only does this reduce the conceptual gap for programmers, which makes it easier to implement and reason about protocols, but by preserving all relevant context information, high-level protocol specifications also offer considerable novel opportunities for compilers to do optimizations on multicore hardware.

Additionally, Reo has several software engineering advantages as a domain-specific language for protocols [7]. For instance, Reo forces developers to separate their computation code from their protocol code. Such a separation facilitates verbatim reuse, independent modification, and compositional construction of protocol implementations (i.e., connectors) in a straightforward way. Moreover, Reo has a strong mathematical foundation [8], which enables formal connector analyses (e.g., model checking [9]). This makes statically verifying deadlock-freedom in a given protocol, for instance, relatively easy; such analyses are much harder to perform on code in lower-level general-purpose languages.

To use connectors for implementing protocols among concurrent processes in real applications, one must derive implementations from their graphical specification, as precompiled executable code or using a run-time interpretation engine. Roughly two implementation approaches currently exist. In the *distributed approach* [10–13], one implements the behavior of each of the k constituents of a connector and runs these k implementations concurrently as a distributed system; in the *centralized approach* [7,14,5,6], one computes the behavior of a connector as a whole, implements this behavior, and runs this implementation sequentially as a centralized system.

In ongoing work, we are developing Reo compilers (Reo-to-Java [7], Reo-to-C [14]) according to the centralized approach based on Reo's formal automata semantics [15]. On input of a graphical connector specification (as an XML file), these tools automatically generate code in four steps. *First*, they extract from the specification a list of the channels and nodes constituting the specified connector. *Second*, they consult a database to find for every channel and node in the list a "small" automaton that formally describes the behavior of that particular channel. *Third*, they compute the product of the automata in the constructed collection to obtain one "big" automaton that describes the behavior of the whole connector. *Fourth*, they feed a data structure representing that big automaton to a template. Essentially, this template is an incomplete fragment of sequential code with "holes" that need be "filled" (with information from the data structure). At run-time, the generated code simulates the big automaton computed in the third step: it runs sequentially in its own *protocol process* and responds to events (i.e., I/O operations) coming from the *computation processes* under coordination that perform the real work (i.e., its environment).

Problem Computing one big automaton (the third step of the centralized approach) and afterward translating it to sequential code (the fourth step) has two problems. First, computing the product may require too many resources at compile-time, because such computations require, in the worst case, time and space exponential in the (state space) size of the largest small automaton. Second, even if our compiler succeeds in computing the big automaton, the subsequently generated *sequential* code may unnecessarily restrict parallelism among independent transitions at run-time[1]: such sequential code serializes independent transitions, forcing them to execute one after the other (see Section 2.2 for details). Consequently, although formally sound, the generated implementation may run overly sequentially (e.g., if the first transition to execute takes a long time to complete, while other transitions could have fired manifold during that time).

One approach to solving these two problems is to *not* compute one big automaton but generate code directly for each of the small automata instead, essentially moving from the centralized approach to the distributed approach: using a distributed algorithm, the implementations of the small automata apply the product operators between them at run-time instead of at compile-time. Although this approach solves the stated problem—independent transitions can execute

[1] Independent transitions cannot disable each other by firing.