Science of Computer Programming 96 (2014) 377–394

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

Constructive polychronous systems ☆

Jean-Pierre Talpin [a,*], Jens Brandt [b], Mike Gemünde [b], Klaus Schneider [b], Sandeep Shukla [c]

[a] INRIA Rennes-Bretagne-Atlantique, France
[b] University of Kaiserslautern, Germany
[c] Virginia Tech, USA

HIGHLIGHTS

- Constructive semantic framework over a complete domain for imperative and declarative synchronous languages.
- First executable small-step operational semantics of polychronous data-flow languages.
- Characterization of program correctness (determinism, endochrony) by fixpoint properties.

ARTICLE INFO

Article history:
Received 14 March 2013
Received in revised form 19 March 2014
Accepted 7 April 2014
Available online 24 April 2014

Keywords:
Synchronous programming
Operational semantics
Constructive systems
Fixpoint theory
Program verification

ABSTRACT

The synchronous paradigm provides a logical abstraction of time for reactive system design which allows automatic synthesis of embedded systems that behave in a predictable, timely, and reactive manner. According to the synchrony hypothesis, a synchronous model reacts to inputs by generating outputs that are immediately made available to the environment. While synchrony greatly simplifies the design of complex systems in general, it can sometimes lead to causal cycles. In these cases, constructiveness is a key property to guarantee that the output of each reaction can still be always algorithmically determined.
Polychrony deviates from perfect synchrony by using a partially ordered, i.e., a relational model of time. It encompasses the behaviors of (implicitly) multi-clocked data-flow networks of synchronous modules and can analyze and synthesize them as GALS systems or Kahn process networks (KPNs).
In this paper, we present a unified constructive semantic framework using structured operational semantics, which encompasses both the constructive behavior of synchronous modules and the multi-clocked behavior of polychronous networks. Along the way, we define the very first executable operational semantics of the polychronous language SIGNAL.

1. Introduction

Languages such as ESTEREL [1], QUARTZ [2] or LUSTRE [3] are based on the *synchrony hypothesis* [4,5]. Synchrony is a logical abstraction of time which greatly facilitates verification and synthesis of safety-critical embedded systems. In particular, it enforces deterministic concurrency, which has many advantages in system design, e.g. avoiding Heisenbugs (i.e. bugs

☆ This work is partially supported by INRIA associate project Polycore, by the Deutsche Forschungsgemeinschaft (DFG), by the US Air Force Research Laboratory (grant FA8750-11-1-0042) and the US Air Force Office for Scientific Research (grant FA8655-13-1-3049).
* Corresponding author.
E-mail addresses: Jean-Pierre.Talpin@inria.fr (J.-P. Talpin), brandt@cs.uni-kl.de (J. Brandt), gemuende@cs.uni-kl.de (M. Gemünde), schneider@cs.uni-kl.de (K. Schneider), shukla@vt.edu (S. Shukla).

http://dx.doi.org/10.1016/j.scico.2014.04.009
0167-6423/

that disappear when one tries to simulate/test them), predictability of real-time behavior, as well as provably correct-by-construction software synthesis [6].

It is also the key to generate *deterministic* single-threaded code from multi-threaded synchronous programs so that synchronous programs can be directly executed on simple micro-controllers without using complex operating systems. Another advantage is the straightforward translation of synchronous programs to hardware circuits [7] that allows one to use synchronous languages in HW/SW co-design. Furthermore, the concise formal semantics of synchronous languages allows one to formally reason about program properties [8], compiler correctness and worst-case execution time [9,10].

Under the synchrony hypothesis, computation progresses through totally ordered synchronized execution steps called reactions. The computation involved in reacting to a particular input combination starts by reading the inputs, computing the intermediate and the output values of the reaction, as well as the next state of the system. Each complete reaction is referred to as a *macro-step* whereas computations during the reaction are called *micro-steps*. A reaction is said to happen at a *logical instant* that abstracts the duration of a reaction to a single point in a discrete totally ordered timeline.

Consequently, and from a semantic point of view – which postulates that a reaction is atomic – neither communication nor computation therefore takes any physical time in a synchronous instant. Even though this zero-time assumption does not correspond to reality, it is where the power of the synchronous abstraction lies: zero delay is compatible to predictability. If (1) the minimum inter-arrival time of two consecutive values on all inputs is long enough, and if (2) all micro-steps in a reaction (macro-step) are executed according to their data dependencies, then the behaviors under the zero-time assumption are the same as the behaviors of the same system in reality.

However, the synchronous abstraction of time also has a drawback: Since outputs are generated in zero-time, and since synchronous systems can typically read their own outputs, there may be cyclic dependencies due to actions modifying their own causes within the same reaction. These issues may lead to programs having inconsistent or ambiguous behaviors. In the context of synchronous programs, they are known as *causality problems*, and various solutions have been proposed over the years to tackle them.

The most obvious and pragmatic one is to syntactically forbid cyclic data dependencies which is simple to check but rules out many valid programs. For example, the synchronous data-flow language Lustre follows this approach [11]. More powerful algorithms for causality analysis are discussed in Section 2. One key advantage of Lustre is that it corresponds to a strict subclass of Kahn networks in which all actors synchronize on every execution step, so called synchronous Kahn networks [12]. Kahn networks are none to have a compositional asynchronous semantics [13].

In contrast to synchronous languages, the polychronous language Signal [14] follows a different model of computation. Execution is not aligned to a totally ordered set of logical instants but to a partially ordered model of time. This allows one to directly express (abstractions of) asynchronous computations which possibly synchronize intermittently. The lack of a global reference of time offers many advantages for the design of embedded software.

First, it is closer to reality since at the system level, integrated components are typically designed based on different clock domains or different paces, which is a desirable feature especially with the advent of, e.g., multi-core embedded processors. Second, polychrony avoids unnecessary synchronization, thereby offering additional optimization opportunities. Polychrony gives developers the possibility to refine the system in different ways, and compilers can choose from different schedules according to non-functional mapping constraints, which are ubiquitous in embedded systems design. Due to these advantages, Signal is particularly suited as a coordination layer on top of synchronous components to describe a globally asynchronous locally synchronous (GALS) network.

As Signal makes use of the synchronous abstraction of time, it faces the same problems as other synchronous languages. One way to overcome the causality problem is to syntactically forbid cyclic dependencies, but as stated before that is not always possible, especially when composing separately specified processes. The Signal compiler uses a so-called *conditional dependence graph* [15–17] to model dependencies between equations and to check that all equations in a syntactic cycle cannot happen at the same logical instant. As discussed above, the synchronous languages are all based on slightly varying notions of causality.

This mismatch makes it unnecessarily hard (if not impossible) to currently integrate, e.g., a set of reactive Quartz modules with a Signal data-flow network: should the integration of modules and processes be limited/approximated by syntactically causal data-flow networks instead of constructive ones? There is no fundamental reason why a common notion of constructiveness should not exist for these languages. So, instead of an approach to causality analysis based on syntactic cycle detection, we want to endow Signal with a constructive semantics compatible to that of languages like Quartz, which is exactly what this paper presents.

2. Related work

As mentioned in the introduction, causality analysis performed in data-flow languages is a conservative approach to checking the constructiveness of a system. However, mapping or composing models on platforms often requires the introduction of pseudo-cycles [18–20]. Therefore, other synchronous languages like Esterel [1] opted for a more sophisticated solution. Their semantics is given in terms of a constructive logic, and compilers perform a causality analysis [19–25] based on the computation of fix-points in a three-valued logic similar to Brzozowski and Seger's ternary simulation of asynchronous circuits [26]. Thereby, cyclic dependencies are allowed as long as they can be constructively resolved. This