Foreword to the special section on synchronous programming

Jean-Pierre TALPIN

INRIA Rennes-Bretagne-Atlantique, Rennes 35042, France

© Higher Education Press and Springer-Verlag Berlin Heidelberg 2013

1 Introduction

Embedded systems are ubiquitous. In applications ranging from avionics, automotive, and industrial process control all the way to the handheld personal digital assistants (PDAs), cell phones, and bio-medical prosthetic devices, we find embedded computing devices running embedded software systems.

Growth in embedded system design is exponential. There are orders of magnitude more embedded processors and embedded applications today in deployment than other forms of computing. Some of the applications running on such embedded computing platforms are also safety-critical, real-time, and require absolute guarantees of correctness, timeliness, and dependability.

The design of such safety-critical applications require the utmost care. These applications must be verified for functional correctness; satisfaction of real-time constraints must be ensured; and must be properly endowed with reliability/dependability properties. These requirements pose hard challenges to system designers. A large body of research done over the last few decades exists in the field of designing safety-critical hardware and software systems.

A number of standards have evolved, specifications have been published, architecture analysis and design languages have been proposed, techniques for optimal mapping software components to architectural elements have been studied, modular platform architectures have been standardised.

Certification by various authorities and strict requirements for satisfying certification goals have been developed.

Today's avionics, automotive, process control, and many other safety-critical systems are usually developed based on such a large body of knowledge and technology.

Yet, the primary goal of safety-critical embedded system design remains to produce an implementation that meets customer requirements and safety standards imposed by certification authorities. At the same time, development and maintenance costs must be kept under control and possibly meet performance, power, weight constraints as well as physical limitations imposed by the system architecture itself (e.g., a physical distribution of electronic control units (ECUs), sensors and actuators).

Meeting these objectives demands design methods and tools that integrate seamlessly with existing design flows and are built on solid mathematical foundations. To meet such entangled and conflicting objectives, confidence in the design is paramount. Anything outside the designers' practice, knowledge, experience is bound to reduce that confidence. Mathematics is not.

Using a solid mathematical foundation to build a safety-critical system provides the designer with the ability to formally and unambiguously reason about the operations of the system. Certification is simplified because ambiguity is out of the way and makes it possible to formally prove the correct operation of the system. It also improves productivity as part of the implementation can now be automatically generated so as to meet some of the requirements (determinism, schedulability, etc.).

Towards achieving the goal of designing mathematically well-founded safety-critical systems, especially embedded software in this way, a number of disciplines have been developed, and are being researched at the moment. These research areas consist of programming models, abstractions,

semantics preserving refinements, real-time and other non-functional constraint based model elaboration, automated synthesis of code, and formal verification.

2 The synchronous paradigm

In Europe, a number of programming models and languages based on such programming models have evolved for this purpose. Synchronous and polychronous programming models constitute important classes among them. Stream based programming models, Petri nets, process algebra and various semantically enriched UML models also belong to these.

In the US, actor based modelling paradigms, tuple-space based modelling, and guarded command languages are the main ones, other than automata based composition of processes.

As early as in the 80s, in particular, several French laboratories developed domain-specific design languages for embedded software design, so called synchronous languages, all based on some common fundamental features:

1. Concurrency — To support functional concurrency and provide an intuitive way to compose elementary blocks: dataflow diagrams, hierarchical automata, imperative programs, or any notation suitable to the targeted engineering community.
2. Simplicity — To possess the simplest formalisation possible to make reasoning tractable, comencing with the semantics of composition.
3. Synchrony — To support the simple and frequently-used model where all reactions of the system are assumed to take finite memory and time.

Combining synchrony and concurrency while maintaining a simple mathematical model is not straightforward. Synchrony divides time into discrete instants. This model is pervasive in mathematics and engineering.

It appears in automata, in the discrete-time dynamical systems familiar to control engineers, and in synchronous digital logic familiar to hardware designers. Hence it is natural to assume that a synchronous program progresses according to successive atomic reactions.

However, there is no free lunch. Composition requires forming the conjunction of reactions of a system's components. It is well known that such a conjunction will not in general be a function but rather a relation or, equivalently, a

constraint.

The product of automata is a familiar example: two automata must synchronise when performing a shared transition; this is a constraint. The same occurs in hardware: composing two systems does not always produce a valid circuit because it is possible to create a delay-free (i.e., cyclic) between each of the components.

This problem of synchronous systems not being closed under composition has been addressed in at least four different ways: defining microsteps (e.g., VHDL, StateCharts), enforcing cycle-freedom (e.g., Lustre), checking existence of unique fixed-points (Esterel) or solving systems of relations constraints, which is the case of the Signal language, of more particular focus in the present volume.

Still, defining a programming model, its semantics, and translation schemes from high-level notations into execution languages such as C is only one part of the task. Another important one is to create a proper design methodology around the language, programming model, and support it with tools for creating executable software from it.

3 A special section

Following up upon these works and seminal results, the synchronous paradigm has been further developed across many boundaries of embedded system design and applied in then unsuspected areas of research.

This special issue of the journal on *Frontiers of Computer Science* aims at providing a glance on current research issues related to synchronous programming, from formally verified code generation, distributed code generation, applications to architecture exploration and simulation and to formally control variability in embedded software engineering, to formal semantics.

The first article, by Ngo et al., is entitled “Formal verification of synchronous data-flow program transformations toward certified compilers”. Translation validation is a technique introduced in the 90's to formally verify the correctness of code generators. Rather than certifying the code generator itself or exhaustively qualifying it w.r.t. DO-330 standards, translation validation provides a scalable approach to assess the functional correctness of the generated code by simply verifying semantic preservation from one source specification to its automatically generated code. Ngo et al. give an extensive address to the formalisation of program transformations performed on Signal program prior to actual C code generation, and on instrumenting each of these steps with for-

mal refinement relations amenable to automated verification using SMT (satisfiability modulo theory)-solving.

The second article, by Hu et al., is entitled “Multithreaded code generation from signal program to OpenMP”. It investigates advanced automated code generation techniques for the synchronous language Signal targeting multicore execution platforms. The authors propose a methodology to translate Signal programs into OpenMP-based multi-threaded C code supporting deployment on multi-core architectures.

The third article, by Yu et al., is entitled “Exploring system architectures in AADL via POLYCHRONY and SYNDEX”. Architecture analysis & design language (AADL) has been increasingly adopted in the design of embedded systems, and corresponding scheduling and formal verification have been studied extensively. The present contribution pinpoints some of the less explored issues of instrumenting a specification formalism like the AADL with real-time distributed code generation techniques to support its use for architecture exploration and real-time simulation. This application of the is based on a translation of the AADL into Signal which allows to use the toolset of the Polychrony environment for the that particular purpose of simulating virtual hardware platforms.

The fourth article, by Millo et al., is entitled “Scenario-based verification in presence of variability using a synchronous approach”. This contribution proposes a new model of scenarios, dedicated to verifying the behaviour and supporting the variability of software product lines. The authors use the model of computations and communication of the synchronous language Esterel in order to capture abstractions of system behaviours and conduct a formal verification methodology applied to the composition and parameterisation of software components.

The fifth article, by Yang et al., is entitled “A comparative study of two formal semantics of the SIGNAL Language”. It focuses on the data-ow synchronous language Signal and gives an updated account to its semantics by considering its

trace semantics, its tagged semantics and proving equivalence relations between them using the automated proof assistant Coq. This study yields the definition of an intermediate, pivot, semantic model which allows to divide the problem from a large and ambitious proof and conquer two simpler equivalence proofs.

4 Closing note

This volume is intended to provide readers with a sample on advanced topics and state of the art in various fields related to synchronous programming. Hence, it is obviously not an exhaustive handbook. Our hope is that the reader will find the area of research covered in this selection important and of interest, and further the study of this field of research in related topics from the provided references and from other sources.

Jean-Pierre Talpin is a Senior Researcher with INRIA and leads the project team who develops the open-source Polychrony environment. He received his PhD from Université Paris VI Pierre et Marie Curie in 1993. He then was a research associate with the European Computer-Industry Research Centre in Munich before to join INRIA

in 1995. Jean-Pierre edited two books with Elsevier and Springer, guest-edited more than ten special issues of ACM and IEEE scientific journals, and authored more than 20 journal articles and book chapters and 60 conference papers. He received the 2004 ACM Award for the most influential POPL paper, for his 2nd conference paper with Mads Tofte, and the 2012 LICS Test of Time Award, for his 1st conference paper with Pierre Jouvelot.