Hindawi Publishing Corporation
EURASIP Journal on Embedded Systems
Volume 2007, Article ID 32903, 2 pages
doi:10.1155/2007/32903

Editorial
Synchronous Paradigm in Embedded Systems

Alain Girault,[1] S. Ramesh,[2] and Jean-Pierre Talpin[1]

[1] *INRIA, France*
[2] *IIT Bombay, India*

Received 19 June 2007; Accepted 19 June 2007

Synchronous languages were introduced in the 1980s for programing reactive systems. Such systems are characterized by their continuous reaction to their environment, at a speed determined by the latter. Reactive systems include embedded control software and hardware. Synchronous languages have recently seen a tremendous interest from leading companies developing automatic control software and hardware for critical applications. Industrial success stories have been achieved by Schneider Electric, Airbus, Dassault Aviation, Snecma, MBDA, Arm, ST Microelectronics, Texas Instruments, Freescale, Intel, and so on. The key reason for the success is in the rigorous mathematical semantics provided by the synchronous approach that allows programers to develop critical software and hardware in a faster and safer way.

Indeed, an important feature of synchronous paradigm is that the tools and environments supporting development of synchronous programs are based upon a formal mathematical model defined by the semantics of the languages. The compilation involves the construction of these formal models, and their analysis for static properties, their optimization, the synthesis of executable sequential implementations, and the automated distribution of programs. It can also build a model of the dynamical behaviors, in the form of a transition system, upon which is based the analysis of dynamical properties, for example, through model-checking-based verification, or discrete controller synthesis. Hence, synchronous programing is at the cross-roads of many approaches in compilation, formal analysis and verification techniques, and software or hardware implementations generation.

We invited for this special issue of the journal original papers on all aspects of the synchronous paradigm for embedded systems, including theory and applications. We received initially 9 papers for the special issue. After the first round of reviews, 7 papers were short-listed or recommended for major revision or resubmission. All the authors came back with resubmissions which were subjected to an additional round of rigorous reviews, which resulted in the 5 papers that appear in this special issue.

In the paper titled "A domain-specific language for multitask systems, applying discrete controller synthesis" by G. Delaval and E. Rutten, a programing language for multitasking real-time control systems called Nemo is proposed. Nemo specifies a control layer, on top of the computation layer underlying an embedded application. The idea is to specify a set of resources with usage constraints, a set of modes of each task with its resource requirements, and a set of applications sequencing the tasks, as well as some global temporal constraints on task interactions, so that an application specific task handler can be generated automatically, obeying all the constraints. The task handler is nothing but an application specific scheduler. It really specifies when to start and stop tasks, so as to meet the control objectives by staying within specified resource constraints. It is generated using the technology of optimal control synthesis for discrete event systems.

In the paper "Removing cycles in Esterel programs," the authors J. Lukoschus and R. von Hanxleden present a new algorithm to transform a cyclic but causal Esterel program (hence constructive) into an equivalent acyclic one. The transformation is performed at the source level. Technically, the detection of the cyclic dependencies is performed thanks to a structural induction on the Esterel source program, while the cycle elimination is performed by iteratively replacing signals involved in cycles by signal expressions involving other signals not involved in cycles. This algorithm has been implemented in the Columbia Esterel Compiler, and the authors have extensively tested it against other Esterel compilers (namely V5, and V7, with and without the usual optimizations). The results show that most of the signals added by the

transformation are in fact removed afterwards by the Esterel compiler.

In the paper "Code generation in the Columbia Esterel compiler," by S. A. Edwards and J. Zeng, the authors present the three main code generation algorithms for Esterel, based, respectively, on program dependence graphs (PDG), dynamic lists (DL), and a virtual machine (VM). This presentation is very clear and pedagogical. The PDG-based code generation takes advantage of the possibility to reorder statements to remove unnecessary dependencies, therefore avoiding as much context switching as possible. The DL-based code generation tries to avoid generating code for portions of programs that will not run in the future. The VM-based code generation attempts to minimize the code size by employing an Esterel-dedicated virtual machine. The authors have compared the size and speed of the code generated by the three algorithms with that generated by the V3, V5, and V7 compilers. The VM-based code is the slowest while the PDG-based code is the fastest. Regarding the size, the VM-based code is the smallest for the large Esterel programs, while the DL-based code is the largest, but never as big as the V5 or V3 generated code.

In the paper "Array iterators in lustre: from a language extension to its exploitation in validation," by L. Morel, the author presents an extension of the synchronous dataflow language lustre, with array iterators allowing a simple and efficient way to write programs that manipulate arrays. The extension has been designed as a compromise between expressivity and ability to be compiled into reasonable sequential code: an iterator leads to a for-loop in the target language. This work covers all the aspects traditionally covered by Lustre: code generation and proof. A proof environment has been prototyped by the author to assist the user in the application of the proposed methodology. The presented extensions have been successfully applied on an industrial case study and have been adopted by the tool provider for the forthcoming release of Scade 6.

In the paper "Formal methods for scheduling of latency-insensitive designs," by J. Boucaron, R. de Simone, and J.-V. Millo, the authors address the design of latency-insensitive architectures. They provide a solution for balancing the latencies on paths in a circuit under design. This is done by analyzing the design using weighted event/marked graph, where the computation nodes are shown as vertices, and latencies are marked on the arcs. Relay stations are inserted based on the latencies. To ensure that the graph is equalized, new elements called fractional registers are inserted on the faster arc to slow them down. These fractional registers are mostly used in graphs with nodes that have loops. Their operation is statically scheduled based on the flow of tokens through the path. The description of the relay station and shell-wrapper circuitry is provided.

ACKNOWLEDGMENTS

We thank the authors for submitting their work for this special issue. We are indebted to the reviewers who spent considerable time to thoroughly review the papers and help improve the quality of presentations and the issue. We also thank the Editor-in-Chief for encouraging us to bring out this special issue.

Alain Girault
S. Ramesh
Jean-Pierre Talpin