Formally Unifying Modeling and Design for Embedded Systems - A Personal View

G. Berry(✉)

Collège de France, 11 Place Marcelin Berthelot, 75005 Paris, France
gerard.berry@college-de-france.fr
http://www-sop.inria.fr/members/Gerard.Berry

Abstract. Based on the author's academic and industrial experience, we discuss the smooth relation between model-based design and programming realized by synchronous languages in the embedded systems field. These languages are used to develop high quality embedded software, in particular for safety-critical applications in avionics, railway, etc., subject to the strongest software certification processes in industry. They have also been used for the efficient model-based development of production hardware circuits. One of their main characteristics is their well-defined formal semantics, with is the base of their simulation and compiling processes and is also fundamental for their link to automatic formal verification systems and other tools related to model-based design. We briefly discuss their current limitations and some ideas to lift them.

1 Introduction

A unified and preferably formal path for modeling, design, development and verification of circuits and programs is an old dream of many researchers in the embedded systems community. After years of scientific progress and experimentation, the dream slowly becomes a reality. Formal modeling, design and verification methods are now considered as serious ways to build dependable systems, at least by some part of industry. There are several reasons for this relatively recent change. First, the quest for new formal languages, design methods and verification tools has given positive results in the form of well-founded, well-designed and industry-usable development systems. Second, bugs are really not welcome for safety-critical embedded systems in avionics, railways, etc., nor for mission-critical systems such as rockets and satellites; in some famous cases, the cost induced by a single bug has exceeded the cost of the whole development. This motivates industry to try other solutions than traditional manual coding and testing. Third, the most serious certification processes (e.g., DO-178C avionics certification) have now officially recognized the value of formal modeling and design in the certification process, which used to be mostly administrative.

Nevertheless, except for a few integrated methods, the formal landscape remains quite scattered and difficult to understand for most industry engineers. The community should now recognize that is time to present the subject and the achievements in a more organized way, stressing its strengths and recognizing its weaknesses. This paper is a contribution to this goal, based on the author's

T. Margaria and B. Steffen (Eds.): ISoLA 2016, Part II, LNCS 9953, pp. 134–149, 2016.
DOI: 10.1007/978-3-319-47169-3_11

40 years of research and development in Academia and Industry, notably through the experience of the Esterel [12] and SCADE synchronous languages and their applications in both software and hardware industrial projects.

2 The Modeling and Design Landscape

As an activity, programming is quite easy to define: one write texts or graphics that are compiled into some machine language and executed by some computer. Modeling is not as clear-cut, because it deals with many more concepts and objects. One can model the needs of a customer, an information flow, an architecture, the intended executable application, its execution environment, its users, etc. Here, we restrict our attention to *formal modeling*, based on mathematics and computer science concepts and techniques. Formal (or semi-formal) modeling is often of great help to understand, dimension, design, and verify systems. It actually existed much before computer science, being a standard activity in physics or mechanics. In Informatics, the situation is quite contrasted: model-based design is the rule in some application domains, e.g., avionics, and still quite rare in others, e.g. hardware circuits design.

2.1 Integrated Vs. Toolbox-Based Views

The engineering needs are multiple for embedded hardware and software: architectural and microarchitectural design and modeling, precise specification, program or circuit development, verification, integration in the final system, and maintenance during the system's lifetime. These needs are quite different, use different mindsets and tools, and are usually fulfilled by different people. There are roughly two main views to address them.

In the *integrated view* of model-based design, everything is done in a single formalism to which is applied a number of strongly connected tools. Good examples for embedded software are Abrial's B [2] and Event-B [3] methods, where the modeling, specification and actual programming are all done in the B or Event-B set-theoretical languages. Integrated design and verification tools such as Atelier B and the Rodin platform [1] make it possible to formally verify properties of specifications, refine abstract specifications into concrete ones in a formally verified way, and generate embedable code from the concrete specifications. These methods have been successfully applied to industrial systems, for instance automatic subways in Paris and other towns worldwide, and more generally railway signaling. The advantage is of course the full control and homogeneity of the whole chain. The drawback may be a form of rigidity implied by the unique language and some difficulty to absorb local progress made by other theories and tools.

On the opposite, in the *toolbox* view, each step is done with a specific tool, all tools being linked together by a global IDE (Integrated Development Environment). Then the languages, tools, and verification methods in the toolsets may be developed independently of each other, possibly by several universities and

companies. This is by far the dominant model. When the tools are developed and presented in coherent way, preferably using common interchange formats, and when they are well-integrated by the IDE, the development chain is felt by the user as a unified design chain; SCADE Suite by Esterel Technologies is a good example. The advantage is flexibility, the difficulty is to maintain global coherence and correctness of the tools and of their mutual interfaces.

2.2 The Hardware Design Case

In hardware design, the CAD path from ideas to circuits is long and complex. It is a typical toolset-based path. It involves a large number of languages and tools: for specification, mostly text/graphics documents and C/C++/SystemC prototypes; for high-level modeling and simulation-based verification, ISA (Instruction Set Architecture) definitions and simulators for microprocessors, and transaction-level models for SoCs (Systems on Chips) written for instance in SystemC/TLM (Transaction Level Modeling, IEEE standard 1666); for programming, actually called *design* in this field, hardware description languages such as Verilog and VHDL; for design testing, random/directed test generation using hardware verification languages such as e [40]; for design verification, temporal logic formalisms such as PSL (Property Specification Languages, IEEE standard 1850) dealt with by various simulators and model-checkers, or other formal tools dedictated to explicit or symbolic execution trajectory evaluation; for low-level synthesis, gate-level languages with fancy Boolean gate sizing and optimization algorithms based on Binary Decision Diagrams (BDDs) [28,42]. Furthermore, the correctness of most transformations can be formally verified using Boolean satisfiability (SAT) solvers [43], etc. This path is highly complex and uses lots of software tools linked by heavy scripts. Nevertheless, the results are remarkably solid.

A weakness is that most hardware-oriented languages still have informal and sometimes quite fuzzy semantics. In addition, they were designed originally with simulation in mind, not synthesis. This may lead to unexpected difficulties, especially when comparing simulation and synthesis. But formal methods do appear in a growing number of verification steps: property verification of models and designs, verification of all logic optimization steps, equivalence of designs before and after transistor-level synthesis, etc.

2.3 The Safety-Critical Software Case

Safety-critical software plays a major role in several engineering fields: avionics, where it was actually born long ago due to the impossibility for humans to pilot unstable airplanes, the space industry, where satellites vitally depend on software, railways and subways, which have a long tradition of caring with safety, nuclear plants, heavy industry, etc. These domains are submitted to quite stringent certification processes that now recognize the difference between software and mechanics. The most elaborate one is the DO-178C avionics software international standard (see also the DO-254 avionics hardware standard).

Some other domains unfortunately do not yet consider themselves at the same level of criticality. Automotive is a good example, as it seems that the very nature of software is not well understood by many of its actors who still speak of *electronics* and concentrate on cost-reduction more than quality. Development cost is a real economic concern since a certified software is indeed much more expensive than a hastily written one, but it should be compared with the cost of bugs for users and the company. Recent major and lethal problems encountered by a Japanese company with an engine control design that may spontaneously put the engine full speed and by German and American companies with major security flaws allowing to open cars or even to take almost full control of them from the Internet are illustrative examples; for the latter case, it should never be forgotten that security issues most always result from design flaws or apparently innocuous non-functional bugs. Another potential example is medical appliances: I have personally no idea of how and by whom the software of a pacemaker or a robot surgeon is verified, and I have seldom met doctors aware of the problem. And, in several application areas, I have seen companies starting R&D evaluation of safety and security issues by hiring PhD students; they will certainly evaluate the PhD student, but not necessarily the issues.

2.4 Continuous and Discrete Control

Critical embedded software is often related to continuous, discrete or mixed control, and thus on Control Theory. Continuous control is critical to fly an airplane, regulate an engine, or control the brakes and suspension of a car. Any continuous control model must involve a description of a controller at some abstraction level, a model of the physics of the device to control, and a model of the environment. The tools used there are mostly mathematical modelers such as Matlab/Simulink, Modelica, or their competitors, used by engineers trained in Control Theory. Discrete control is critical for airplane cockpits, communication protocols, robot actions control, etc. The typical modeling and implementation tools are based on finite-state machines formalisms that can be simple, hierarchical, concurrent with many possible form of communication, etc. A difficulty is that continuous and discrete control do require quite different skills and thus training. Mixed control appears when both forms of control appear together, for instance when an airplane switches between a number of different flight modes.

Discrete and mixed control are definitely not places where classical mathematical modeling excels, to say the least. Some modelers use hierarchical state machines graphical formalisms with the right drawings but horrendous semantics. And most modelers exhibit strange behaviors when dealing with cascades of discrete events during their basic time-based integration process: they keep relying on incremental integration techniques to handle discrete events, which means that time continues advancing even if causal event cascades should take conceptually no time [8]. It is then possible to see balls traversing walls, for instance.

3 Personal Experience with Formal Modeling and Programming

3.1 The Formal Synchronous Languages

I have worked on formal methods since the beginning of the 1970s and more specifically on embedded systems since 1982. Most of my work has concerned the development of a new way to program embedded systems with *synchronous concurrency* [9,14,37] instead of the asynchronous concurrency that was the mandatory paradigm at that time in Computer Science. Synchronous concurrency simply assumes that computation is defined by a temporal sequence of timeless discrete reactions to external events (or clock ticks, or whatever you like), where computing the reaction to input event and communicating between concurrent processes take no time. Another equivalent way of thinking is that reactions to events are instantaneoulsy computed by a conceptually infinitely fast machine. This idea is not novel: when writing a discretized continuous control equation $z_t = x_t + y_t$ in Control Theory, one always neglects the time it takes to perform + and =. At run-time, one needs of course to check that the physical reaction time is reasonable w.r.t. application constraints, for example by relying on WCET (Worst Case Execution Time) computation tools such as aiT by AbsInt[1]. Similarly, in the Register Transfer Level (RTL) view of a synchronous digital circuit, the cascade of actions that occur during a clock cycle is conceptually seen as instantaneous, while the physical timing closure computation performed by the electronic CAD system ensures that the final voltages of the circuit wires are as defined by the RTL equations at the end of the clock cycle. This greatly simplifies design and verification, since one deals with synchronous discrete Boolean equations instead of asynchronous voltage propagation.

Unlike asynchronous concurrency, synchronous concurrency is deterministic by construction, which makes it very natural for many applications in digital circuit design, continuous and discrete control, robotics, man/machine interface, etc., which are inherently both concurrent and deterministic. An interesting fact is that engineers trained in Control Theory understand and adopt synchrony immediately, which is not the case for most engineers trained in Computer Science with the idea that concurrency is synonym to asynchrony. This clearly shows that modeling and programming are definitely a question of scientific culture.

The three initial synchronous languages were Esterel [12,19,27] for discrete control flow, and Lustre [38] by P. Caspi and N. Halbwachs and Signal [36] by A. Benveniste and P. Le Guernic for continuous control and signal processing. They were developed in interdisciplinary labs gathering researchers in Computer Science and Control Theory, all on fully formal grounds and aimed at industrial applications. They have indeed all become industrial.

At about the same time, the Statecharts [39] graphical formalism was developed for discrete control by D. Harel, with similar ideas but technically quite different semantics. Its great ideas of hierarchical and concurrent graphical state

[1] www.absint.com.

machines were soon borrowed by the synchronous community to develop graphical versions of the synchronous languages such as SyncCharts [4] for Esterel, Argos [48] for Lustre, the Sildex IDE developed by the TNI company for Signal, as well as the MARTE UML profile[2]. Statecharts also served as the basis for the Statemate industrial product[3], the various state machine designs of UML, the Mathwork Stateflow product, etc.

The more recent synchronous languages such as SCADE 6 by Esterel Technologies [29] are hybrids of these initial models, with the addition of a bunch of new ideas that appeared later. All their industrial developments have involved developing tools ranging from code generation to automatic test generation and formal verification. The temptation to adopt half-baked constructs with half-baked semantics to please some particular user has always been resisted: it is very easy to kill the mathematical and practical consistency of a language by such constructs. This was felt as bad scientific taste by the authors, and, more importantly definitely unacceptable for safety-critical applications.

Other stable academic synchronous languages are ReactiveC [26] by F. Boussinot, which embeds Esterel's ideas into C, Reactive ML [47] by L. Mandel and M. Pouzet, which does the same for Caml, and Lucid Synchrone [30] by M. Pouzet *et al.*, which is a higher-order functional synchronous language. More recently defined, SCL [62] by R. Van Hanxleden *et al.* is a direct extension of C with constructive synchronous threads that relaxes Esterel constraints, ScCharts [61] is a version of SyncCharts based on SCL, HipHop [20] by M. Serrano and myself is an Esterel-based extension of the Scheme-based HOP system [57] dedicated to Web programming and orchestration, HipHop-js by C. Vidal plays the same role for the Hop-js [58] javascript version of Hop, and the ideas of Esterel have been embedded in the new algorithmic music score definition language of the Antescofo system [31,32] by A. Cont *et al.* for real-time human/computer music based on adaptive score following. From the points of view of modeling and programming, there is actually not much difference between programming an airplane or an electronic orchestra.

3.2 Synchronous Languages : Modeling or Programming?

Conventional programming languages remain mentally close to the structure of the computer. On the contrary, the synchronous languages try to remain as close as possible to the structure of the problem to be solved; they hierarchically describe abstract temporal behaviors instead of concretely specifying machine instructions to execute. Would it be appropriate to also call them modeling languages?

In a sense yes, since their programming style mostly reflects previously existing modeling activities. For instance, to define the Lustre [38] synchronous programming language, the control theorist P. Caspi studied the way control engineers write airplane control models; Lustre was then developed

[2] http://www.omg.org/spec/MARTE/1.1/PDF/.
[3] http://www-03.ibm.com/software/products/en/ratistat.

with N. Halbwachs, a computer scientist, precisely with the goal of blurring the distinction between modeling and programming. Because it was both simpler and more powerful, Airbus finally preferred SAGA [10], the industrial graphical version of Lustre, to its own internally developed programming language SAO. This lead to the industrial SCADE (Safety Critical Application Development Environment) product.

In an other sense no, because synchronous languages are deterministic and fully executable, which is not mandatory for other modeling activities and may limit specification power. Technically speaking, to help higher-level modeling, one can introduce non-determinism in synchronous languages by using external "oracle" signals acting as drivers for asynchrony. In some cases, it is quite natural, but it may be artificial in others (but see the Averest project at http://www.averest.org for a formal integration attempt of synchronous and asynchronous behavior). We do not have enough rooms to further analyze this question here.

Statecharts were also explicitly designed as a modeling formalism to help the discussion between airplane engineers and pilots. When designing them, D. Harel was looking for the maximal expressive power, not for direct implementability. But the design was good enough to be also almost directly implementable. In its industrial version and in its appropriation by synchronous languages, several restrictions have been used to make the charts more synchronous without losing much expressivity. Here again, the frontier between modeling and programming is not clear.

Altogether, to classify synchronous languages, I think that it would be fair to view them as model-level programming languages that do generate embedded code - I mean real code that actually pilots many modern airplanes and controls their engines, brakes, displays, etc., or does similar things for many other critical functions in many other critical industrial systems.

4 The Evolution of Esterel and SCADE

4.1 Esterel v5, from Research to Industry (1982–2000)

The Esterel language was developed at Ecole des Mines and Inria Sophia-Antipolis from 1982 to 2000. The language style was initiated by two control theory researchers, J.-P. Rigault and J.-P. Marmorat, again extending and systematizing ideas of time-related discrete control modeling [19]. The first formal semantics was given by L. Cosserat and myself in 1984 [16], and the first Esterel v2 compiler was written by P. Couronné and myself in 1985 based on this semantics. G. Gonthier developed novel ideas in his seminal work on efficient semantics [17] that lead a bigger group to implement the much more efficient compiler Esterel v3 from 1986 on. This academic compiler produced C code and also input for the Auto/Autograph verification system [53] based on process-calculi bisimulation techniques. It was readily used for industrial R&D projects, especially for avionics discrete control modeling and formal verification for the Rafale fighter at Dassault Aviation [15] (testing system, landing gear control, cockpit GUI, etc.), for telecommunication at Bell Labs [50], AT&T [41] and British Telecom,

and for robot control at Inria [34]. In the latter case, it is interesting to note that Esterel served as the target language of a robotics domain-specific task description language that provided higher-level modeling based on domain knowledge. The translation to Esterel made it possible to translate robotics models to C and to perform formal verification on them.

But a strong practical limitation was that the Esterel v3 compiler generated deterministic state machines that could and sometimes did explode exponentially in size.

A major progress occurred in 1989–1990, when I worked with J. Vuillemin's hardware group at the Digital Equipment Paris Research Lab. They were developing the Perle programmable FPGA-based board [21] using the first really usable Xilinx FPGAs (programmable circuits). They were very smart in designing fancy data path circuits, but much less at developing the control circuits that drive them. After having tried the well-known one-hot hardware implementation of the automata generated by the v3 compiler, we discovered a much more direct and efficient compiling technique to translate Esterel programs to circuits in a quasi-linear way [11]. The resulting v4 compiler solved once for all the generated code explosion problem. It was readily incorporated in the Agel IDE for Esterel sold by ILOG.

Then, together with H. Touati, J.C. Madre and O. Coudert at Digital Equipment and E. Sentovich and H. Toma at UC Berkeley and Inria, we developed BDD-based optimizers for the generated circuits with excellent practical results [55,56,60], rapidly followed by the Xeve BDD-based formal verifier [24] developed by A. Bouali and R. de Simone at Inria. In practice, our optimized control circuits proved systematically smaller, faster, and easier to verify than human-designed ones. The main reason is that the Esterel modeling style naturally leads to a very efficient, scalable and optimizable state assignment, which is a key for sequential circuits timing and verification efficiency. Another reason is that human beings seem quite incapable of directly designing efficient control circuits with the usual lower-level languages, unlike for data paths.

We could readily adapt the new Esterel v4 hardware compiler to generate C software by simply simulating the circuit in C. Later on, S. Edwards and then D. Potop wrote very different compilers to C [33,51] that generate much more efficient C code that can be either used for circuit simulation or embedded within software systems. The technique and the generated code are quite different, but, thanks to the formal semantics of the language, the results are behaviorally equivalent.

But Esterel v4 did not accept all the programs formerly accepted by Esterel v3, because it was limited to circuits with acyclic combinational structure. This was not a strong limitation for hardware since most circuit CAD tools reject combinational cycles (although S. Malik showed in [46] that cyclic circuits can be more natural and space-efficient than acyclic ones) and since Lustre and most data-flow languages also reject cycles. But our avionics software partners found it natural to program with behaviorally correct combinational cycles; such cycles happen to be cut at some place during each execution step, for instance

by an and-gate receiving a 0 from a wire not in the cycle, but not at the same place for all executions steps [15]. The problem was to find which cyclic circuits should be considered as correct, knowing that equations such as "$X = X$" and "$X =$ not X" had to be rejected. Esterel v3 had heuristics for that, but not quite complete ones; we had to solve the problem in a better way. In 1991, extending S. Malik's seminal work [46] with T. Shiple and H. Touati [59], we characterized the circuits that correctly behave for all values of wire and gate delays: they are exactly those whose equations can be solved by Constructive Boolean Logic, i.e., Boolean logic without the excluded middle law "Xor not $X = true$" instead of classical logic. The typical counter-example is the amazing *Hamlet* circuit "$ToBe = ToBe$ or not $ToBe$" that cannot be solved without using the excluded middle law, which is not available in constructive logic. This circuit never computes $false$, computes $true$ for some wire and gate delays, but does oscillate for some other delays. The initial complicated proof has been recently simplified and made elegant by M. Mendler [49] using a temporal logic of analog stabilization of voltages in circuits, which closes the field at least for Esterel needs.

The Esterel semantics has been unchanged since then, see [13]. More importantly, all the aforementioned semantics remained fully equivalent on the programs they handle in common. We never had to change the language nor the semantic principles.

In 1995, Esterel v5 was also integrated in the Cocentric System Studio tool developed in the US by Synopsys for system-level hardware design, a nascent form of model-based design for circuit design and hardware/software codesign. It was also made part of Cadence's Polis [7] system for hardware/software codesign. But the industry was not yet ready for these design levels and success was meager.

4.2 Esterel v7 for Hardware Design (2001–2009)

At the end of the 1990s, the improved hardware translation of Esterel raised the interest of major actors of the circuit industry, mainly Intel, Xilinx, Texas Instruments, ST micro-electronics, and NXP (formerly Philips). See [18] for instance. But Esterel v5 was weak in data handling. Together with M. Kishinevsky from Intel Strategic CAD Lab in Portland, we developed a much richer version Esterel v7 of the language[4], which enriched the control-flow constructs and added powerful data manipulation constructs. In addition, a novel arithmetic type system allowed us to optimally implement bit-level sizing of variables and communication signals, automatizing a classical headache in data path sizing. The resulting language was very powerful for joint data path and control path handling, both handled at a much higher temporal modeling level than with conventional HDLs.

At the Esterel Technologies company, created in 2000, the Esterel v7 compiler was incorporated into a rich IDE called Esterel Studio that covered design,

[4] http://www.inria.fr/members/Gerard.Berry/papers/Esterelv7ReferenceManual7.60.pdf.

simulation with symbolic debugging, formal verification, synthesis, and documentation of circuits. The software compiler generated C and SystemC circuit simulation code and was directly linked with the Prover SL verifier of Prover Technologies to perform SMT (Satisfaction Modulo Theories) formal verification [35,44], test generation, and construction of counter-examples for dissatisfied formulae. A fast-C code generator based on the aforementioned work by S. Edwards and D. Potop [51] was then implemented to improve generated C code performance. The hardware synthesizer and optimizer of Esterel v5 was also improved and coupled with data path circuit generation; it generated standard VHDL or Verilog. Strangely enough, it took us a lot of time to ensure that VHDL/Verilog simulation and logic synthesis of our quite trivial generated code exactly agreed, although this was stated as "obvious for the synthesizable designs" by CAD tools vendors. Finally, circuit synthesis was made modular to improve the optimization of very large designs.

Around 2005, because of the evolution of SoCs (Systems on Chips) towards multiple clock support and dynamic frequency regulation to save power, Esterel v7 was extended to support clock gating (a key to power saving) and multiclock designs [6]. Surprisingly, this did not require any change to the Esterel mathematical semantics, but only the addition of a new *weak suspension* statement previously introduced by K. Schneider in his Quartz language [54]. These extensions provided our users with the first model-level behavioral view of multiclock design and verification.

After various R&D experimentation successes, Esterel v7 entered in production in 2006 at Texas Instruments for the design of various tricky IP blocks such as smart memory controllers, DMAs (Direct Memory Access units), a hardware decoder for full HD TV on smartphones, etc., and for NoC (Network-on-Chip) design at ST Microelectronics. These designs were made at a much higher level than with classical HDLs, verified early in the loop, and did synthesize excellent hardware.

Industrial practice obliged us to deal with something we never heard of in research: *ECOs*, i.e., *Engineering Change Orders* [5]. This strange name depicts the following situation. When the first samples of a circuit come back from factory, bugs are found that have escaped the extensive simulation and verification campaign. These bugs most often concern tricky control paths such as memory access control, functioning mode logic, or communication protocols, exactly where Esterel v7 was beneficial and used. Usually, such bugs were easy to fix on the source Esterel v7 code. But it would be too long and too expensive to completely rebuild the circuit masks: recompiling and resynthesizing the source code as standard for software is not a possibility for hardware. The bugs must be corrected by *patching the masks*, as traditionally done for printed circuit boards; this was non-trivial since the logic out of Esterel v7 program is very heavily optimized. We first had to make our combinational and sequential optimizations *reversible*, i.e., to make it possible to reconstruct any part of the source logic from the mask. Fortunately, this did not affect much optimization quality. Then, using the source-to-circuit traceability mechanism we had put in place for

symbolic debugging, we could find ways to appropriately patch the masks and formally prove behavioral equivalence between the source change and the patch. In production integrated circuit design, if you cannot do that, you cannot play.

There were other interesting surprises. For instance, when a design is sufficiently advanced, it is sent to and external test team in charge of comparing it to the paper specification and finding its bugs. Such a team is rated by the number of bugs it finds, according to accumulated experience. External testing teams for the Esterel v7-based projects found almost no bugs in the designs, became misjudged because they were rated by the number of bugs found, and bitterly complained about that new state of affairs! It was not easy to convince program managers that it is a good idea to find bugs *before* testing the designs (we did not try to convince the test team it could be smaller).

Unfortunately, the 2008 financial crisis hit massively the circuit industry and reduced severely the number of designs teams. Esterel Technologies had to abandon the development of Esterel v7 and commit to SCADE for certified software. The Esterel Studio software now belongs to Synopsys, which has put it in the deep freezer, and the ongoing IEEE standardization process with academic and industrial partners has been also abandoned. Sigh...

4.3 From Lustre/SCADE to SCADE 6 for Safety-Critical Software

Since synchronous languages are at ease with both hardware and software because their technical problems are similar enough, our initial plan at Esterel Technologies was to attack both markets with Esterel v7. But this turned out to be difficult since the industrial traditions and thus the selling arguments were completely different in both domains. Fortunately, we could buy SCADE from Telelogic in 2003. We then decided to attack the software market with a new product called SCADE 6 [29], whose language unifies the best features of SCADE for data flow and Esterel/SyncCharts for control, while adding support for functional arrays that had become indispensable in industrial applications. The resulting language is defined by its formal semantics, not by words. As for the previous SCADE systems, the SCADE 6 code generator (written in CAML) is DO-178B qualifiable as a development tool, which greatly simplifies software certification of applications and recertifcation after changes.

SCADE Suite is a complete IDE with simulation, formal verification, a qualifiable display generator, links to mathematical modeling, links with SYSML modeling for architectural engineering, etc. Many other tools are linked to SCADE: a translator of Simulink designs; Astrée [23], a fancy abstract interpretation verifier developed by P. Cousot and his team with Airbus to verify generate code properties, and in particular check absence of possible run-time errors; and the StackAnalyzer and aiT abstract-interpretation based tools developed by the AbsInt company to verify stack size compliance and computer WCET (Worst Case Execution Time).

SCADE Suite is used by more than 250 customers worldwide for all kinds of safety-critical software applications. I think it can be viewed as a good example

of technical unification of model-based design and programming within a precise application domain.

5 Open Issues in Model-Based Embedded Systems Design

In theory there is no difference between theory and practice. In practice there is. (Yogi Berra)

Even in the specific domain we discussed, there are many issues to solve to really unify model-based design and programming at both theoretical and practical levels, practical achievements being the real success criterion at the end of the day. I will only cite some of them here, related to currently weak points of the design chain.

Most mathematical modelers for differential equation simulation still lack solid semantics, and, as said before, do not correctly support the mixture of continuous control and discrete event handling. An elegant theoretical solution to continuous/discrete cooperation has been proposed by Benveniste, Bourke, Caillaud, and Pouzet [8]. It is based on non-standard analysis: in addition to progressing by real ϵ's between integration steps, time can progress by *infinitesimal* ϵ's during discrete event cascades. More practically, Pouzet and his team are defining the Zelus simulation language and compiler [25,52], with a type-checker that sorts out continuous and discrete behaviors to ensure that simulation behavior exactly respects the semantics that mathematically defines the expected system behavior. Such a language could advantageously replace the existing ones in mathematical modelers and solve the current continuous/discrete conflicts.

Most code generation tools end up generating C code. But C compilers are not as robust as one usually thinks. For instance, using smart random generation techniques, the CSmith project has generated one million C programs especially triggered to shake C compilers. CSmith found lots of bugs in most tested compilers, be them academic or industrial. These bugs can be compiler crashes or internal errors, which is harmless, but they can also be wrong generated code, which is really harmful and raises questions about the "certification by large usage" often invoked in industry. Only one compiler survived: CompCert [45] by Xavier Leroy and his team. This is not surprising since CompCert has been developed and formally verified with Coq [22], much of its code being automatically extracted from the proof. Such a formally verified and reasonably efficient compiler should definitely be used for safety-critical systems.

Following the same track, L. Rieg and myself are currently feeding Coq with the chain of (Kernel) Esterel semantics up to circuit translation, with the hope of constructing a Coq-verified compiler - and to finally publish my draft book "The Constructive Semantics of Pure Esterel"[5] with all currently unpublished proofs of the theorems done in Coq. Similarly, T. Bourke and others are working on a Coq-verified Lustre compiler.

[5] http://www-sop.inria.fr/members/Gerard.Berry/Papers/EsterelConstructiveBook.pdf.

Finally, and most importantly, the models described in this paper correspond to compact 20^{th}-century embedded systems, to which the core synchronous framework is well-suited. But the embedded systems zoo of the 21^{st} century has many more animals, and in particular physically distributed systems mixing signal processing, complex control, fancy GUIs, etc. There have been many attempts to automatically distribute the code generated by synchronous languages (not detailed here), but more general ways to tackle the problem should be investigated.

Ptolemy II[6], developed at UC Berkeley by Edward Lee's team, is an exciting and elegant environment for model-based design of distributed systems. Instead of being based on a single computation paradigm, Ptolemy II supports a variety of computation and communication models, including the synchronous one, and links them quite cleanly within a global graphical framework. I think such a system can play a major role in the unification of model-based design and programming. Other extensions of the synchronous paradigm are the aforementioned SCL approach [62] and the Averest project by K. Schneider *et al.* (http://www.averest.org).

6 Conclusion

We have shown the direct and formal connection between model-based design and programming in the synchronous languages framework, and detailed its industrial tooling and applications developments in the embedded systems application area. By adopting a higher-level model-based way of writing and verifying designs, we could simplify and put closer modeling and programming *in this application domain*. There are many other places where unification of modeling and programming should be performed, probably in a different way. Isola will be an excellent occasion of discussing this.

References

1. Rodin Users Handbook. http://www3.hhu.de/stups/handbook/rodin/current/html/
2. Abrial, J.R.: The B-book: Assigning Programs to Meanings. Cambridge University Press, New York (1996)
3. Abrial, J.R.: Modeling in Event-B: System and Software Engineering. Cambridge University Press, New York (2013)
4. André, C.: Representation, analysis of reactive behaviors: a synchronous approach. In: Proceedings of CESA 1996, IEEE-SMC, Lille, France (1996)
5. Arditi, L., Berry, G., Kishinevsky, M.: Late design changes (ECOs) for sequentially optimized Esterel designs. In: Proceedings of Formal Methods in Computer Aided Design, FMCAD 2004, Austin, Texas (2004)
6. Arditi, L., Berry, G., Kishinevsky, M., Perreaut, M.: Clocking schemes in Esterel. In: Proceedings of Designing Correct Circuits, DCC 2006, Vienna, Austria (2006)

[6] http://ptolemy.eecs.berkeley.edu/ptolemyII/.

7. Balarin, F., Chiodo, M., Jurecska, A., Hsieh, H., Lavagno, A.L., Passerone, C., Sangiovanni-Vincentelli, A., Sentovich, E., Suzuki, K., Tabbara, B.: Hardware-Software Co-Design of Embedded Systems: The Polis Approach. Kluwer Academic Press (1997)
8. Benveniste, A., Bourke, T., Caillaud, B., Pouzet, M.: Non-standard semantics of hybrid systems modelers. J. Comput. Syst. Sci. (JCSS) **78**(3), 877–910 (2012). Special issue in honor of Amir Pnueli
9. Benveniste, A., Caspi, P., Edwards, S., Halbwachs, N., Le Guernic, P., de Simone, R.: The synchronous languages 12 years later. Proc. IEEE **91**(1), 64–83 (2003)
10. Bergerand, J.L., Pilaud, E., Saga,: a software development environment for dependability in automatic control. In: Proceedings of Safecomp 1988. Pergamon Press (1988)
11. Berry, G.: A hardware implementation of pure Esterel. Sadhana Acad. Proc. Eng. Sci. Indian Acad. Sci. **17**(1), 95–130 (1992)
12. Berry, G.: The foundations of Esterel. In: Proof, Language and Interaction Essays in Honour of Robin Milner. MIT Press (2000)
13. Berry, G.: The Constructive Semantics of Pure Esterel. Draft book version 3 (without proofs) (2002). http://www-sop.inria.fr/members/Gerard.Berry/Papers/EsterelConstructiveBook.pdf
14. Berry, G., Benveniste, A.: The synchronous approach to reactive and real-time systems. Another Look Real Time Programm. Proc. IEEE **79**, 1270–1282 (1991)
15. Berry, G., Bouali, A., Fornari, X., Nassor, E., Ledinot, E., de Simone, R.: Esterel: a formal method applied to avionic development. Sci. Comput. Program. **36**, 5–25 (2000)
16. Berry, G., Cosserat, L.: The ESTEREL synchronous programming language and its mathematical semantics. In: Brookes, S.D., Roscoe, A.W., Winskel, G. (eds.) CONCURRENCY 1984. LNCS, vol. 197, pp. 389–448. Springer, Heidelberg (1985). doi:10.1007/3-540-15670-4_19
17. Berry, G., Gonthier, G.: The Esterel synchronous programming language: design, semantics, implementation. Sci. Comput. Program. **19**(2), 87–152 (1992)
18. Berry, G., Kishinevsky, M., Singh, S.: System level design and verification using a synchronous language. In: Proceedings of International Conference on Integrated Circuit Design, ICCAD 2003, San Jose, USA (2004)
19. Berry, G., Moisan, S., Rigault, J.-P.: Towards a synchronous and semantically sound high level language for real-time applications. In: IEEE Real Time Systems Symposium, pp. 30–40 (1983). IEEE Catalog 83 CH 1941–4
20. Berry, G., Serrano, M., Hop, H.: Multitier web orchestration. In: Proceedings of the ICDCIT 2014 Conference, pp. 1–13 (2014)
21. Bertin, P., Roncin, D., Vuillemin, J.: Programmable active memories: a performance assessment. In: Borriello, G., Ebeling, C. (eds.) Research on Integrated Systems: Proceedings of the 1993 Symposium, pp. 88–102 (1993)
22. Bertot, Y., Casteran, P.: Interactive Theorem Proving and Program Development-Coq'Art: The Calculus of Inductive Constructions. Springer (2004)
23. Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: A static analyzer for large safety-critical software. In: PLDI 2003 ACM SIGPLAN SIGSOFT Conference on Programming Language Design and Implementation, San Diego, California, USA, pp. 196–207 (2003)
24. Bouali, A.: Xeve: an Esterel verification environment. In: Proceedings of Computer Aided Verification, CAV 1998, Vancouver, Canada (1998)

25. Bourke, T., Colaço, J.-L., Pagano, B., Pasteur, C., Pouzet, M.: A synchronous-based code generator for explicit hybrid systems languages. In: Franke, B. (ed.) CC 2015. LNCS, vol. 9031, pp. 69–88. Springer, Heidelberg (2015). doi:10.1007/978-3-662-46663-6_4
26. Boussinot, F., Reactive, C.: An extension of C to program reactive systems. Softw. Pract. Exp. **21**(4), 401–428 (1991)
27. Boussinot, F., de Simone, R.: The Esterel language. Another Look Real Time Programm. Proc. IEEE **79**, 1293–1304 (1991)
28. Bryant, R.E.: Graph-based algorithms for Boolean function manipulation. IEEE Trans. Comput. **35**(8), 677–691 (1986)
29. Colaço, J.-L., Pagano, B., Pouzet, M.: A conservative extension of synchronous data-flow with state machines. In: Proceedings of Emsoft 2005, New Jersey, USA (2005)
30. Colaço, J.-L., Girault, A., Hamon, G., Pouzet, M.: Towards a higher-order synchronous data-flow language. In :ACM Fourth International Conference on Embedded Software, EMSOFT 2004, Pisa, Italy, September 2004
31. Cont, A.: A coupled duration-focused architecture for real-time music-to-score alignment. IEEE Trans. Pattern Anal. Mach. Intell. **32**, 974–987 (2010)
32. Echeveste, J., Cont, A., Giavitto, J.-L., Jacquemard, F.: Operational semantics of a domain specific language for real time musician-computer interaction. Discrete Event Dyn. Syst. **23**(4), 343–383 (2013)
33. Edwards, S.: An Esterel compiler for large control-dominated systems. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **2**(2), 169–183 (2002)
34. Espiau, B., Coste-Manière, E.: A synchronous approach for control sequencing in robotics applications, pp. 503–508. In: Proceedings of IEEE International Workshop on Intelligent Motion, Istambul (1990)
35. De Moura, L., Bjrner, N.: Satisfiability modulo theories: introduction and applications. Comm. ACM **54**(9), 69–77 (2011)
36. Le Guernic, P., Le Borgne, M., Gauthier, T., Le Maire, C.: Programming real time applications with Signal. Another Look Real Time Programm. Proc. IEEE **79**, 1270–1282 (1991). Special Issue
37. Halbwachs, N.: Synchronous Programming of Reactive Systems. Kluwer, Dordrecht (1993)
38. Halbwachs, N., Caspi, P., Pilaud, D.: The synchronous dataflow programming language Lustre. Another Look Real Time Programm. Proc. IEEE **79**, 1270–1282 (1991). Special Issue
39. Harel, D.: Statecharts: a visual approach to complex systems. Sci. Comput. Program. **8**, 231–274 (1987)
40. Iman, S., Joshi, S.: The e-Hardware Verification Language. Springer, Heidelberg (2004)
41. Jagadeesan, L., Von Olnhausen, J., Puchol, C.: A formal approach to reactive system software: a telecommunications application in Esterel. J. Formal Methods Syst. Des. **8**(2), 132–145 (1996)
42. Knuth, D.: The Art of Computer Programming, Vol. 4: Combinatorial Algorithms, Section 7.1.4: Binary Decision Diagrams. Addison Wesley, Reading (2014)
43. Knuth, D.: The Art of Computer Programming, vol. 4B, 7.2.2.2: Satisfiability. Addison Wesley, Reading (2016)
44. Kroening, D., Strichman, O.: Decision Procedures An Algorithmic Point of View. Springer (2008)
45. Leroy, X.: Formal verification of a realistic compiler. Commun. ACM **52**(7), 107–115 (2009)

46. Malik, S.: Analysis of cyclic combinational circuits. IEEE Trans. Comput. Aided Des. **13**(7), 950–956 (1994)
47. Mandel, L., Pouzet, M.: ReactiveML, a reactive extension to ML. In: Proceedings of Principles and Practice of Declarative Programming, PPDP 2005, Lisbon (2005)
48. Maraninchi, F., Rémond, Y.: Mode automata: a new domain-specific construct for the development of safe critical systems. Sci. Comput. Programm. **46**(3), 219–254 (2003)
49. Mendler, M., Shiple, T., Berry, G.: Constructive Boolean circuits and the exactness of timed ternary simulation. Formal Methods Syst. Des. **40**(3), 283–329 (2012)
50. Murakami, G., Sethi, R.: Terminal call processing in Esterel. In: Proceedings of IFIP 92 World Computer Congress, Madrid, Spain (1992)
51. Potop-Butucaru, D., Edwards, S.A., Berry, G.: Compiling Esterel. Springer, Heidelberg (2007)
52. Pouzet, M.: Building a hybrid systems modeler on synchronous languages principles. In: Proceedings of ACM International Conference on Embedded Software (EMSOFT), Amsterdam (2015)
53. Roy, V., de Simone, R.: Auto and autograph. In: Kurshan, R. (ed.) Proceedings of Workshop on Computer Aided Verification, New-Brunswick, June 1990
54. Schneider, K.: Embedding imperative synchronous languages in interactive theorem provers. In: Proceedings of Conference on Application of Concurrency to System Design (ACSD) (2001)
55. Sentovich, E., Toma, H., Berry, G.: Latch optimization in circuits generated from high-level descriptions. In: Proceedings of International Conference on Computer-Aided Design (ICCAD) (1996)
56. Sentovich, E., Toma, H., Berry, G.: Efficient latch optimization using exclusive sets. In: Proceedings of Digital Automation Conference (DAC) (1997)
57. Serrano, M., Berry, G.: Multitier programming in Hop - a first step toward programming 21st-century applications. Commun. ACM **55**(8), 53–59 (2012)
58. Serrano, M., Prunet, V.: A glimpse of Hopjs. In: 21th Sigplan International Conference on Functional Programming (ICFP), Nara, Japan (2016)
59. Shiple, T., Berry, G., Touati, H.: Constructive analysis of cyclic circuits. In: Proceedings of International Design and Testing Conf (ITDC), Paris (1996)
60. Touati, H., Berry, G.: Optimized controller synthesis using Esterel. In: Proceedings of International Workshop on Logic Synthesis IWLS 1993, Lake Tahoe (1993)
61. von Hanxleden, R., Duderstadt, B., Motika, C., Smyth, S., Mendler, M., Aguado, J., Mercer, S., OBrien, O.: SCCharts: Sequentially constructive statecharts for safety-critical applications. In: Proceedings ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI14), Edinburgh, UK, (2014)
62. von Hanxleden, R., Mendler, M., Aguado, J., Duderstadt, B., Fuhrmann, I., Motika, C., Mercer, S., O'Brien, O.: Sequentially constructive concurrency - a conservative extension of the synchronous model of computation. In: Proceedings of Design, Automation and Test in Europe Conference, DATE 2013, Grenoble, France (2013)