The Synchronous Hypothesis and Synchronous Languages

Dumitru Potop-Butucaru[*] IRISA, Rennes, France

Robert de Simone INRIA, Sophia-Antipolis, France

Jean-Pierre Talpin IRISA, Rennes, France

November 2, 2004

Abstract

The design of electronic Embedded Systems relies on a number of different engineering disciplines. As the domain becomes ever more important, both in theoretical challenges and in industrial relevance, it has drawn a considerable attention in recent years, with a number of proposals for formalisms, languages and models to help support the design flow. In this article we shall describe Synchronous Reactive languages, which emerged as early as the 1980's decade, and are now gaining increasing recognition for their modeling adequacy to embedded systems.

1 Introduction

Electronic Embedded Systems are not new, but their pervasive introduction in ordinary-life objects (cars, phones, home appliances) brought a new focus onto design methods for such systems. New development techniques are needed to meet the challenges of productivity in a competitive environment. This handbook reports on a number of such innovative approaches to the matter. We shall concentrate here on Synchronous Reactive (S/R) languages [34, 10, 4, 7].

S/R languages rely on the *synchronous hypothesis*, which lets computations and behaviors be divided into a discrete sequence of *computation steps* which are equivalently called *reactions* or *execution instants*. In itself, this assumption is rather common in practical embedded system design. But the synchronous hypothesis adds to this the fact that, *inside each instant*, the behavioral propagation is well-behaved (causal), so that the status of every signal or variable is established and defined prior to being tested or used. This criterion, which may be seen at first as an isolated technical requirement, is in fact the key point of the approach. It ensures strong semantic soundness by allowing universally recognized mathematical models such as the Mealy machines and the digital circuits to be used as supporting foundations. In turn, these models give access to a large corpus of efficient optimization, compilation, and formal verification techniques. The synchronous hypothesis also guarantees full equivalence between various levels of representation, thereby avoiding altogether the pitfalls of non-synthesizability of other similar formalisms. In that sense, the synchronous hypothesis is, in our view, a major contribution to the goal of *model-based design* of embedded systems.

Structured languages have been introduced for the modeling and programming of S/R applications. They are roughly classified in two families:

imperative languages, such as Esterel [13, 20, 14] and SyncCharts [2], provide constructs to shape control-dominated programs as *hierarchical synchronous automata*, in the wake of the StateCharts formalism, but with a full-fledged treatment of simultaneity, priority, and absence notification of signals in a given reaction. Thanks to this, signals assume a consistent status for all parallel components in the system at any given instant.

declarative languages, such as Lustre[35] and Signal[9], shape applications based on intensive data computation and *data-flow organization*, with the control flow part operating under the form of (internally generated) activation clocks. These clocks prescribe which data computation blocks are to be performed as part of the current reaction. Here again, the semantics of the languages deal with the issue of behavior consistency, so that every value needed in a computation is indeed available at that instant.

[*]partly supported by the ARTIST IST European project

We shall describe here the Synchronous Hypothesis and its mathematical background, together with a range of design techniques empowered by the approach and a short comparison with neighboring formalisms; then we introduce both classes of S/R languages, with their special features and a couple of programming examples; finally we comment on the benefits and shortcomings of S/R modeling, closing with a look at future perspectives and extensions.

2 The Synchronous Hypothesis

2.1 What for?

Program correctness (the process performs as intended) and program efficiency (it performs as fast as possible) are major concerns in all of computer science, but they are even more stringent in the embedded area, as no on-line debugging is feasible, and time budgets are often imperative (for instance in multimedia applications).

Program correctness is sought by introducing appropriate syntactic constructs and dedicated languages, making programs more easily understandable by humans, as well as allowing high-level modeling and associated verification techniques. Provided semantic preservation is ensured down to actual implementation code, this provides reasonable guarantees on functional correctness. However, while this might sound obvious for traditional software compilation schemes, the hardware synthesis process is often not "seamless", as it includes manual rewriting.

Program efficiency is traditionally handled in the software world by algorithmic complexity analysis, and expressed in terms of individual operations. But in modern systems, due to a number of phenomena, this "high-level" complexity reflects rather imperfectly the "low-level" complexity in numbers of clock cycles spent. In the hardware domain, one considers various levels of modeling, corresponding to more abstract (or conversely more precise) timing account: transaction-level, cycle-accurate, time-accurate.

One possible way (amongst many) to view Synchronous Languages is to take up the analogy of *cycle-accurate programming* to a more general setting, including (reactive) software as well. This analogy is supported by the fact that *simulation* environments in many domains (from scientific engineering to HDL simulators) often use lockstep computation paradigms, very close to the synchronous cycle-based computation. In these settings, cycles represent logical steps, not physical time. Of course timing analysis is still possible afterwards, and in fact often simplified by the previous division into cycles.

The focus of synchronous languages is thus to allow modeling and programming of systems where cycle (computation step) precision is needed. The objective is to provide domain-specific structured languages for their description, and to study matching techniques for efficient design, including compilation/synthesis, optimization, and analysis/verification. The strong condition insuring the feasibility of these design activities is the *synchronous hypothesis*, described next.

2.2 Basic notions

What has come to be known as the Synchronous Hypothesis, laying foundations for S/R systems, is really a collection of assumptions of a common nature, sometimes adapted to the framework considered. We shall avoid heavy mathematical formalization in this presentation, and defer the interested reader to the existing literature, such as [4, 7]. The basics are:

Instants and reactions: behavioral activities are divided according to (logical, abstract) *discrete time*. In other words, computations are divided according to a succession of non-overlapping *execution instants*. In each instant, input signals possibly occur (for instance by being sampled), internal computations take place, and control and data are propagated until output values are computed and a new global system state is reached. This execution cycle is called the *reaction* of the system to the input signals. Although we used the word "time" just before, there is no real physical time involved, and instant durations need not be uniform (or even considered !). All that is required is that reactions converge and computations are entirely performed before the current execution instant ends and a new one begins. This empowers the obvious conceptual abstraction that computations are infinitely fast ("instantaneous", "zero-time"), and take place only at discrete points in (physical) time, with no duration. When presented without sufficient explanations, this strong formulation of the Synchronous Hypothesis is often discarded by newcomers as unrealistic (while, again, it is only an abstraction, amply used in other domains where "all-or-nothing" transaction operations take place).

Signals: broadcast signals are used to propagate information. At each execution instant, a signal can either be *present* or *absent*. If present, it also carries some value of a prescribed type ("pure" signals exists as well, that carry only their presence status). The key rule is that a signal must be consistent (same present/absent status, same data) for all read operations during any given instant. In particular, reads from parallel components must be consistent, meaning that signals act as controlled shared variables.

Causality: the crucial task of deciding whenever a signal can be declared *absent* is of utter importance in the theory of S/R systems, and an important part of the theoretical body behind the Synchronous Hypothesis. This is of course especially true of *local* signals, that are both generated and tested inside the system. The fundamental rule is that the presence status and value of a signal should be defined *before* they are read (and tested). This requirement takes various practical forms depending on the actual language or formalism considered, and we shall come back to this later. Note that "before" refers here to causal dependency in the computation of the instant, and not to physical or even logical time between successive instants [12]. The Synchronous Hypothesis ensures that *all* possible schedules of operations amount to the same result (convergence); it also leads to the definition of "correct" programs, as opposed to ill-behaved ones where no causal scheduling can be found.

Activation conditions and clocks: Each signal can be seen as defining (or generating) a new clock, ticking when it occurs; in hardware design, this is called *gated clocks*. Clocks and sub-clocks, either external or internally generated, can be used as control entities to activate (or not) component blocks of the system. We shall also call them *activation conditions*.

2.3 Mathematical models

If one forgets temporarily about data values, and one accepts the duality of present/absent signals mapped to true/false values, then there is a natural interpretation of synchronous formalisms as synchronous digital circuits at schematic gate level, or "netlists" (roughly RTL level with only Boolean variables and registers). In turn, such circuits have a straightforward behavioral expansion into Mealy FSMs.

The two slight restrictions above are not essential: the adjunction of types and values into digital circuit models has been successfully attempted in a number of contexts, and S/R systems can also be seen as contributing to this goal. Meanwhile, the introduction of clocks and presence/absence signal status in S/R languages departs drastically from the prominent notion of *sensitivity list* generally used to define the simulation semantics of Hardware Description Languages (HDLs).

We now comment on the opportunities made available through the interpretation of S/R systems into Mealy machines or netlists.

netlists: we consider here a simple form, as Boolean equation systems defining the values of wires and Boolean registers as a Boolean function of other wires and previous register values. Some wires represent input and output signals (with value `true` indicating signal presence), others are internal variables.
This type of representation is of special interest because it can provide exact dependency relations between variables, and thus the good representation level to study *causality* issues with accurate analysis. Notions of "constructive" causality have been the subject of much attention here. They attempt to refine the usual crude criterion for synthesizability, which forbids cyclic dependencies between non-register variables (so that a variable seems to depend upon itself in the same instant), but does not take into account the Boolean interpretation, nor the potentially reachable configurations. Consider the equation $x = y \vee z$, while it has been established that y is the constant `true`. Then x does not *really* depend on z, since its (constant) value is forced by y's. Constructive causality seeks for the best possible faithful notion of true combinatorial dependency taking the Boolean interpretation of functions into account. For details, see [52].
Another equally important aspect of the mathematical model is that a number of combinatorial and sequential optimization techniques have been developed over the years, in the context of hardware synthesis approaches. The main ones are now embedded in the SIS and MVSIS optimization suites, from UC Berkeley [50, 30]. They come as a great help in allowing programs written in high-level S/R formalisms to compile into efficient code, either software or hardware-targeted [51].

Mealy machines: Mealy machines are finite-state automata corresponding strictly to the synchronous assumption. In a given state, provided a certain input valuation (a subset of *present* signals), the machine reacts by immediately

producing a set of output signals before entering a new state.
The Mealy machines can be generated from netlists (and by extension from any S/R system). The Mealy machine construction can then be seen as a symbolic expansion of all possible behaviors, computing the space of reachable states (RSS) on the way. But while the precise RSS is won, the precise causal dependencies relations are lost, which is why Mealy FSM and netlists models are both useful in the course of S/R design [55]. When the RSS is extracted, often in symbolic BDD form, it can be used in a number of ways: We already mentioned that constructive causality only considers dependencies inside the RSS; similarly, all activities of model-checking formal verification, and test coverage analysis are strongly linked to the RSS construction [18, 17, 27, 3].

The modeling style of netlists can be extrapolated to block-diagram networks, often used in multimedia digital signal processing, by adding more types and arithmetic operators, as well as activation conditions to introduce some amount of control-flow. The *declarative* synchronous languages can be seen as attempts to provide structured programming to compose large systems modularly in this class of applications, as described in section 4. Similarly, *imperative* languages provide ways to program in a structured way hierarchical systems of interacting Mealy FSMs, as described in section 3.

2.3.1 Synchronous Hypothesis *vs.* neighboring models

Many quasi-synchronous formalisms exist in the fields of embedded system (co-)simulation: the simulation semantics of SystemC and regular HDLs at RTL level, or the discrete-step Simulink/Stateflow simulation, or the official State-Charts semantics for instance. Such formalisms generally employ a notion of physical time in order to establish when to start the next execution instant. Inside the current execution instant, however, *delta-cycles* allow zero-delay activity propagation, and potentially complex behaviors occur inside a given single reaction. The main difference here is that no causality analysis (based on the Synchronous Hypothesis) is performed at compilation time, so that an efficient ordering/scheduling cannot be pre-computed before simulation. Instead, each variable change recursively triggers further re-computations of all depending variables in the same reaction.

2.4 Implementation issues

The problem of implementing a synchronous specification mainly consists in defining the step *reaction function* that will implement the behavior of an instant, as shown in figure 1. Then, the global behavior is computed by iterating this function for successive instants and successive input signal valuations. Following the basic mathematical interpreta-

```
reaction () {
     decode_state ; read_input ;
     compute ;
     write_output ; encode_state ;
}
```

Figure 1: The reaction function is called at each instant to perform the computation of the current step

tions, the compilation of a S/R program may either consist in the expansion into a flat Mealy FSM, or in the translation into a flat netlist (with more types and arithmetic operators, but without activation conditions). The run-time implementation consists here in the execution of the resulting Mealy machine or netlist. In the first case, the automaton structure is implemented as a big top-level switch between states. In the second case, the netlist is totally ordered in a way compatible with causality, and all the equations in the ordered list are evaluated at each execution instant. These basic techniques are at the heart of the first compilers, and still of some industrial ones.

In the last decade fancier implementation schemes have been sought, relying on the use of activation conditions: During each reaction, execution starts by identifying the “truly useful” program blocks, which are marked as “active”. Then only the actual execution of the active blocks is scheduled (a bit more dynamically) and performed in an order that respects the causality of the program. In the case of declarative languages, the activation conditions come in the form of a hierarchy of clock under-samplings – the clock tree, obtained through a “clock calculus” computation performed at compile time (see section 4.3). In the case of imperative formalisms, activation conditions are based on the halting points (where the control flow can stop between execution instants) and on the signal-generated (sub-)clocks (see section 3.3).

3 Imperative style: Esterel and SyncCharts

For control-dominated systems, comprising a fair number of (sub-)modes and macro-states with activity swapping between them, it is natural to employ a description style that is algorithmic and imperative, describing the changes and progression of control in an explicit flow. In essence, one seeks to represent hierarchical (Mealy) Finite State Machines (FSM), but with some data computation and communication treatment performed inside states and transitions. Esterel provides this in a textual fashion, while SyncCharts propose a graphical counterpart, with visual macro-states. It should be noted that systems here remain finite-state (at least their control structure).

3.1 Syntax and structure

Esterel introduces a specific `pause` construct, used to divide behaviors into successive instants (reactions). The `pause` statement excepted, control is flowing through sequential, parallel and if-then-else constructs, performing data operations and interprocess signaling. But it stops at `pause`, memorizing the activity of that location point for the next execution instant. This provides the needed atomicity mechanism, since the instant is over when all currently active parallel components reach a `pause` statement.

The full Esterel language contains a large number of constructs that facilitate modeling, but there exists a reduced kernel of primitive statements (corresponding to the natural structuring paradigms) from which all the other constructs can be derived. This is of special interest for model-based approaches, because only primitives need to be assigned semantics as transformations in the model space. The semantics of the primitives are then combined to obtain the semantics of composed statements. Figure 2 provides the list of primitive operators for the *data-less* subset of Esterel (also called Pure Esterel). A few comments are here in order:

- in p`;` q the reaction where p terminates is the same as the reaction where q starts (control can be split into reactions only by `pause` statements inside p or q).
- the `loop` constructs do not terminate, unless aborted from above. This abortion can be due to an *external* signal received by an `abort` statement, or to an *internal* exception raised through the `trap/exit` mechanism, or to any of the two (like for the `weak abort` statement). The body of a loop statement should not instantly terminate, or else the loop will unroll endlessly in the same instant, leading to divergence. This is checked by static analysis techniques. Finally, loops are the only means of defining iterating behaviors (there is no general recursion), so that the system remains finite-state.
- the `present` signal testing primitive allows an `else` part. This is essential to the expressive power of the language, and has strong semantic implications pertaining to the Synchronous Hypothesis. It is enough to note here that, according to the synchronous hypothesis, signal *absence* can effectively be asserted.
- the difference between "`abort` p `when` S" and "`weak abort` p `when` S" is that in the first case signal S can only come from outside p and its occurrence prevents p from executing during the execution instant where S arrives. In the second case, S can also be emitted by p, and the preemption occurs only after p has completed its execution for the instant.
- technically speaking, the `trap/exit` mechanisms can emulate the `abort` statements. But we feel that the ease of understanding makes the latter worth inclusion in the set of primitives. Similarly, we shall sometimes use "`await` S" as a shorthand for "`abort loop pause end when` S", and "`sustain` S" for "`loop emit` S `end`".

Most of the data-handling part of the language is deferred to a general-purpose host language (C, C++, Java, . . .). Esterel only declares type names, variables types, and function signatures (which are used as mere abstract instructions). The actual type specifications and function implementations must be provided and linked at later compilation time.

In addition to the structuring primitives of figure 2, the language contains (and requires) interface declarations (for signals, most notably), and modular division with submodules invocation. Submodule instantiation allows signal renaming, *i.e.* transforming virtual name parameters into actual ones (again, mostly for signals). Rather than providing a full user manual for the language, we shall illustrate most of these features on an example.

`[`p`]`	enforces precedence by parenthesis
`pause`	suspends the execution until next instant
p`;` q	executes p, then q as soon as p terminates
`loop` p `end`	iterates p forever in sequence
`[`p `||` q`]`	executes p and q in parallel, synchronously
`signal` S `in end`	declares local signal S in p
`emit` S	emits signal S
`present` S `then` p `else` q `end`	executes p or q upon S being present *or absent !*
`abort` p `when` S	executes p until S occurs (exclusive)
`weak abort` p `when` S	executes p until S occurs (inclusive)
`suspend` p `when` S	executes p unless S occurs
`trap` T `in` p `end`	declare/catch exception T in p
`exit` T	raise exception T

Figure 2: Pure Esterel statements

The small example of fig. 3 has four input signals and one output signal. Meant to model a cyclic computation like a communication protocol, the core of our example is the loop which awaits the input `I`, emits `O`, and then awaits `J` before instantly restarting. The local signal `END` signals the completion of loop cycles. When started, the `await`

```
module Example: input I,J,KILL,SUSP; output O;
suspend
  trap T in %exception handler, performs the preemption
    signal END in
      loop %basic computation loop
        await I;emit O;await J;emit END
      end
    ||
      %preemption protocol, triggered by KILL
      await KILL;await END;exit T
    end
  end;
when SUSP %suspend signal
end module
```

Figure 3: A simple Esterel program modeling a cyclic computation (like a communication protocol) which can be interrupted between cycles and which can be suspended

statement waits for the *next* clock cycle where its signal is present. The computation of all the other statements present in our example is performed during a single clock cycle, so that the `await` statements are the only places where control can be suspended between reactions (they preserve the *state* of the program between cycles). A direct consequence is that the signals `I` and `J` must come in different clock cycles in order not to be discarded.

The loop is preempted by the exception handling statement `trap` when “`exit T`” is executed. In this case, `trap` instantly terminates, control is given in sequence, and the program terminates. The preemption protocol is triggered by the input signal `KILL`, but the exception `T` is raised only when `END` is emitted. The program is suspended – no computation is performed and the state is kept unchanged – in clock cycles where the `SUSP` signal is received. A possible execution trace for our program is given in fig. 4.

3.2 Semantics

Esterel enjoys a full-fledged formal semantics, in the form of *Structural Operational Semantic (SOS)* rules [12]. In fact, there are two main levels of such rules, with the coarser describing all potential, logically consistent behaviors,

clock	inputs	outputs	comments
0	any		all inputs discarded
1	I	O	
2	KILL		preemption protocol triggered
3			nothing happens
4	J,SUSP		suspend, J discarded
5	J		END emitted, T raised, program terminates

Figure 4: A possible execution trace for our example

while the more precise one only selects those that can be obtained in a constructive way (thereby discarding some programs as "unnatural" in this respect). This issue can be introduced with two small examples:

```
     present S then emit S end              present S else emit S end
```

In the first case the signal `S` can logically be assumed as either present or absent: if assumed present, it will be emitted, so it will become present; if assumed absent, it will not be emitted. In the second case, following a similar reasoning, the signal can be neither present, nor absent. In both cases, anyhow, the analysis is done by "guessing" before branching to the potentially validating emissions. While more complex causality paradoxes can be built using the full language, these two examples already show that the problem stems from the existence of causality dependencies inside a reaction, prompted by instantaneous sequential control propagation and signal exchanges. The so-called *constructive causality* semantics of Esterel checks precisely that control and signal propagation are well-behaved, so that no "guess" is required. Programs which pass this requirement are deemed as "correct", and they provide deterministic behaviors for whatever input is presented to the program (which is a desirable feature in embedded system design).

3.3 Compilation and compilers

Following the pattern presented in section 2.4, the first compilers for Esterel were based on the translation of the source into (Mealy) finite automata or into digital synchronous circuits at netlist level. Then, the generated sequential code was a compiled automata or netlist simulator. The automata-based compilation [14] was used in the first Esterel compilers (known as Esterel V3). Automaton generation was done here by exhaustive expansion of all reachable states using symbolic execution (all data is kept uninterpreted). Execution time was then theoretically optimal, but code size could blow up (as the number of states), and huge code duplication was mandatory for actions that were performed in several different states. The netlist-based compilation (Esterel V5) is based on a quasi-linear, structural Esterel-to-circuits translation scheme [11] that ensures the tractability of compilation even for the largest examples. The drawback of the method is the reaction time (the simulation time for the generated netlist), which increases linearly with the size of the program.

Apart from these two previous compilation schemes, which have matured into full industrial-strength compilers, several attempts have been made to develop a more efficient, basically event-based type of compilation which follows more readily the naive execution path and control propagation inside each reaction, and in particular executes "as much as possible" only the truly active parts of the program[1]. We mention here three such approaches: the Saxo compiler of Closse *et al.* [56], the EC compiler of Edwards [28], and the GRC2C compiler of Potop and de Simone [47]. All of them are structured around flowgraph-based intermediate representations that are easily translated into well-structured sequential code. The different intermediate representations also give the differences between approaches, by determining which Esterel programs can be represented, and what optimization and code generation techniques can be applied.

[1]Recall that this is a real issue in Esterel, since programs may contain reaction to *absence* of signals, and determining this absence may require to check that no emission remains possible in the potential behaviors, whatever feasible test branches could be taken. To achieve this goal at a reasonable computational price, current compilers require in fact additional restrictions – in essence, the acyclicity of the dependency/causality graph at some representation level. Acyclicity ensures constructiveness, because any topological order of the operations in the graph gives an execution order which is correct for all instants.

We exemplify on the GRC2C compiler [47], which is structured around the GRC intermediate form. The GRC representation of our example, given in fig. 5, uses two graph-based structures – a *hierarchical state representation* (HSR) and a *concurrent control-flow graph* (CCFG) – to preserve most of the structural information of the Esterel program while making the control flow explicit with few graph-building primitive nodes. The HSR is an abstraction of the syntax tree of the initial Esterel program. It can be seen as a structured data memory that preserves state information across reactions. During each instant, a set of activation conditions (clocks) is computed from this memory state, to drive the execution towards active instructions. The CCFG represents in an operational fashion the computation of an instant (the transition function). During each reaction, the dynamic CCFG operates on the static HSR by marking/unmarking component nodes (subtrees) with "active" tags as they are activated or deactivated by the semantics.

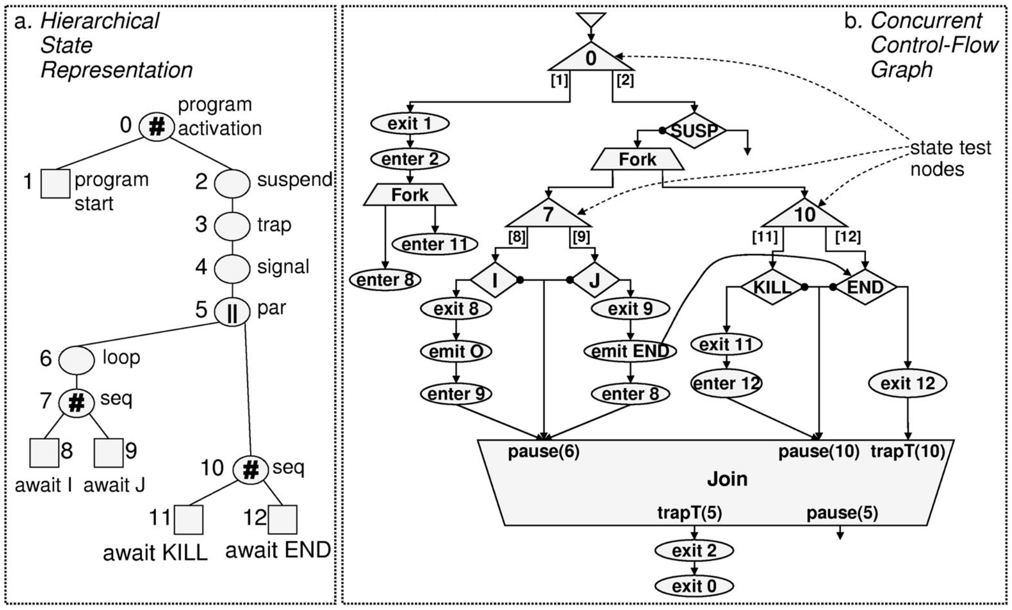

Figure 5: GRC intermediate representation for our Esterel example

For instance, when we start our small example (fig. 3,5), the "program start (1)" and "program (0)" HSR nodes are active, while all the statements of the program (and the associated HSR nodes) are not. Like in any instant, control enters the CCFG by the topmost node and uses the first state decoding node (labelled 0) to read the state of the HSR and branch to the start behavior, which sets the "program start (1)" indicator to inactive (with "exit 1"), and activates "await I" and "await KILL" (with "enter 8" and "enter 11").

The HSR also serves as a repository for tags, which record redundancies between various activation clocks, and are used by the optimization and code generation algorithms. Such a tag is #, which tells that at most one child of the tagged node can retain control between reactions at a time (the activation clocks of the branches are exclusive). Other tags (not figured here) are computed through complex static analysis of both the HSR and CCFG. The tags allow efficient optimization and sequential code generation.

The CCFG is obtained by making the control flow of the Esterel program explicit (a structural, quasi-linear translation process)[2]. Usually, it can be highly optimized using classical compiler techniques and some methods derived from circuit optimization, both driven by the HSR tags computed by static analysis. Code generation from a GRC representation is done by encoding the state on sequential variables, and by scheduling the CCFG operators using classical compilation techniques [43].

The Saxo compiler of Closse *et al.* [56] uses a discrete-event interpretation of Esterel to generate a *compiled event-driven simulator*. The compiler flow is similar to that of VeriSUIF [29], but Esterel's synchronous semantics are used to highly simplify the approach. An *event graph* intermediate representation is used here to split the program

[2]Such a process is necessary, because most Esterel statements pack together two distinct, and often disjoint behaviors: one for the execution instants where they are started, and one for instants where control is resumed from inside.

into a list of *guarded procedures*. The guards intuitively correspond to events that trigger computation. At each clock cycle, the simulation engine traverses the list once, from the beginning to the end, and executes the procedures with an active guard. The execution of a procedure may modify the guards for the current cycle and for the next cycle. The resulting code is slower than its GRC2C-generated counterpart for two reasons: First, it does not exploit the hierarchy of exclusion relations determined by switching statements like the tests. Second, optimization is less effective because the program hierarchy is lost when the state is (very redundantly) encoded using guards.

The EC compiler of Edwards [28] treats Esterel as having control-flow semantics (in the spirit of [40, 43]) in order to take advantage of the initial program hierarchy and produce efficient, well-structured C code. The Esterel program is first translated here into a concurrent control-flow graph representing the computation of a reaction. The translation makes the control flow explicit and encodes the state access operations using tests and assignments of integer variables. Its *static scheduling* algorithm takes advantage of the mutual exclusions between parts of the program and generates code that uses *program counter* variables instead of simple Boolean guards. The result is therefore faster than its Saxo-generated counterpart. However, it is usually slower than the GRC2C-generated code because the GRC representation preserves the state structure of the initial Esterel program and uses static analysis techniques to determine redundancies in the activation pattern. Thus, it is able to better simplify the final state representation and the CCFG.

3.4 Analysis/Verification/Test Generation: Benefits from formal approaches

We claimed that the introduction of well-chosen structuring primitives, endowed with formal mathematical semantics and interpretations as well-defined transformations in the realms of Mealy machines and synchronous circuits, was instrumental in allowing powerful analysis and synthesis techniques as part of the design of synchronous programs. What are they, and how do they appear in practice to enhance the confidence in the correctness of Safety-Critical embedded applications ?

Maybe the most obvious is that synchronous formalisms can fully benefit from the model-checking and automatic verification usually associated to the netlist and Mealy machine representations, and now widely popular in the hardware design community with the PSL/SuGaR and assertion-based design approaches. Symbolic BDD- and SAT-based model-checking techniques are thus available on all S/R systems. Moreover,the structured syntax allows in many cases the introduction of modular approaches, or guide abstraction techniques with the goal of reducing complexity of analysis.

The ability of formal methods akin to model-checking can also be used to automatically produce test sequences which seek to reach the best possible coverage in terms of visited states or exercised transitions. Here again specific techniques were developed to match the S/R models.

Also, symbolic representations of the reachable state spaces (or abstracted over-approximations), which can effectively be produced and certified correct thanks to the formal semantics, can be used in the course of compilation and optimization. In particular for Esterel, the RSS computation allows more “correct” programs w.r.t. constructiveness: indeed causal dependencies may vary in direction depending on the state. If all dependencies are put together regardless of the states, then a causality cycle may appear, while not all components of the cycle may be active at the same instant, and so no real cycle exists (but it takes a dynamic analysis to establish this). Similarly, the RSS may exhibit combinatorial relations between registers encoding the local states, so that register elimination is possible to further simplify the state space structure.

Finally, the domain-specific structuring primitives empowering dedicated programming can also be seen as an important criterion. Readable, easily understandable programs are a big step towards correct programs. And when issues of correctness are not so plain and easy, as for instance when regarding proper scheduling of behaviors inside a reaction to respect causal effects, then powerful abstract hypothesis are defined in the S/R domain that define admissible orderings (and build them for correct programs). A graphical version of Esterel, named SyncCharts for Synchronous StateCharts, has been defined to provide a visual formalism with a truly synchronous semantics.

4 The declarative style: Lustre and Signal

The presentation of declarative formalisms implementing the synchronous hypothesis as defined in Section 2 can be cast into a model of computation (proposed in [33]) consisting of a *domain* of traces/behaviors and of a semi-lattice structure that renders the synchronous hypothesis using a timing equivalence relation: clock equivalence. Asynchrony can be superimposed on this model by considering a flow equivalence relation. Heterogeneous systems [6] can also be modeled by parameterizing the composition operator using arbitrary timing relations.

4.1 A synchronous model of computation

We consider a partially-ordered set of tags t to denote instants (which are seen, in the sense of Section 2.2, as symbolic periods in time during which one reaction takes place). The relation $t_1 \leq t_2$ says that t_1 occurs before t_2. A minimum tag exists, denoted by 0. A totally ordered set of tags C is called a *chain* and denotes the sampling of a possibly continuous or dense signal over a countable series of causally related tags. Events, signals, behaviors and processes are defined as follows:

- an *event* e is a pair consisting of a value v and a tag t,
- a *signal* s is a function from a *chain* of tags to a set of values.
- a *behavior* b is a function from a set of names x to signals.
- a *process* p is a set of behaviors that have the same domain.

In the remainder, we write $\text{tags}(s)$ for the tags of a signal s, $\text{vars}(b)$ for the domains of b, $b|_X$ for the projection of a behavior b on a set of names X and b/X for its complementary. Figure 6 depicts a behavior (b) over three signals named x, y and z. Two frames depict timing domains formalized by chains of tags. Signal x and y belong to the same timing domain: x is a down-sampling of y. Its events are synchronous to odd occurrences of events along y and share the same tags, e.g. t_1. Even tags of y, e.g. t_2, are ordered along its chain, e.g. $t_1 < t_2$, but absent from x. Signal z belongs to a different timing domain. Its tags, e.g. t_3 are not ordered with respect to the chain of y, e.g. $t_1 \not\leq t_3$ and $t_3 \not\leq t_1$.

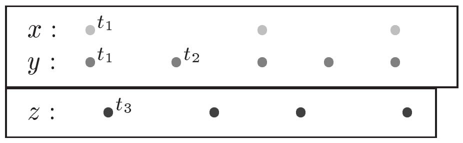

Figure 6: A behavior (named b) over three signals (x, y and z) belonging to two clock domains

The synchronous composition of the processes p and q is denoted $p \,|\, q$. It is defined by the union $b \cup c$ of all behaviors b (from p) and c (from q) which hold the same values at the same tags $b|_I = c|_I$ for all signal $x \in I = \text{vars}(b) \cap \text{vars}(c)$ they share. Figure 7 depicts the synchronous composition, right, of the behaviors b, left, and the behavior c, middle. The signal y, shared by b and c, carries the same tags and the same values in both b and c. Hence, $b \cup c$ defines the synchronous composition of b and c.

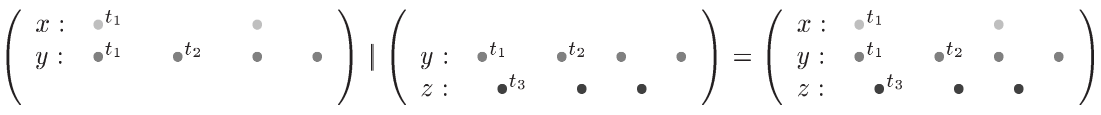

Figure 7: Synchronous composition of $b \in p$ and $c \in q$

A scheduling structure is defined to schedule the occurrence of events along signals during an instant t. A scheduling $\rightarrow$ by a pre-order relation between dates x_t where t represents the time and x the location of the event. Figure 8 depicts such a relation, superimposed to the signals x and y of figure 6. The relation $y_{t_1} \rightarrow x_{t_1}$, for instance, requires y to be calculated before x at the instant t_1. Naturally, scheduling is contained in time: if $t < t'$ then $x_t \rightarrow^b x_{t'}$ for any x and b and it $x_t \rightarrow^b x_{t'}$ then $t' \not\leq t$.

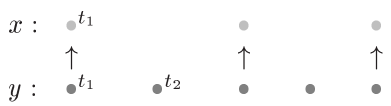

Figure 8: Scheduling relations between simultaneous events

A synchronous structure is defined by a semi-lattice structure to denote behaviors that have the same timing structure. The intuition behind this relation (depicted in figure 9) is to consider a signal as an elastic with ordered marks on it (tags). If the elastic is stretched, marks remain in the same relative and partial order but have more space (time) between each other. The same holds for a set of elastics: a behavior. If elastics are equally stretched, the order between marks is unchanged. In figure 9, the time scale of x and y changes but the partial timing and scheduling relations are preserved. Stretching is a partial-order relation which defines clock equivalence. Formally, a behavior c is a *stretching* of b of same domain, written $b \leq c$, if there exists an increasing bijection on tags f that preserves the timing and scheduling relations. If so, c is the image of b by f. Last, the behaviors b and c are said *clock-equivalent*, written $b \sim c$, *iff* there exists a behavior d s.t. $d \leq b$ and $d \leq c$.

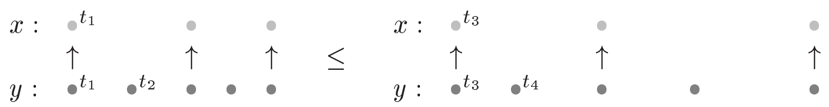

Figure 9: Relating synchronous behaviors by stretching

4.2 Declarative design languages

The declarative design languages Lustre [35] and Signal [9] share the core syntax of figure 10 and can both be expressed within the synchronous model of computation of section 4.1. In both languages, a process P is an infinite loop that consists of the synchronous composition $P \,|\, Q$ of simultaneous equations $x = y\ f\ z$ over signals named x, y, z. Both Lustre and Signal support the restriction of a signal name x to a process P, noted P/x. The analogy stops here as Lustre and Signal differ in fundamental ways. Lustre is a single-clocked programming language, while Signal is a multi-clocked (polychronous) specification formalism. This difference originates in the choice of different primitive combinators (named f in figure 10) and results in orthogonal system design methodologies.

$$P, Q ::= x = y\ f\ z \mid P/x \mid P \,|\, Q$$

Figure 10: A common syntactic core for Lustre and Signal

Combinators for Lustre In a Lustre process, each equation processes the n^{th} event of each input signal during the n^{th} reaction (to possibly produce an output event). As it synchronizes upon availability of all inputs, the timing structure of a Lustre program is easily captured within a single clock domain: all input events are related to a master clock and the clock of the output signals is defined by sampling the master. There are three fundamental combinators in Lustre:

- **Delay:** “$x = \texttt{pre}\, y$” initially lets x undefined and then defines it by the previous value of y.
- **Followed-by:** “$x = y \texttt{->} z$” initially defines x by the value v, and then by z. The `pre` and `->` operators are usually used together, like in “$x = v \texttt{->} \texttt{pre}\,(y)$”, to define a signal x initialized to v and defined by the previous value of y. Scade, the commercial version of Lustre, uses a one-bit analysis to check that each signal defined by a `pre` is effectively initialized by an `->`.

- **Conditional:** "$x = \texttt{if}\, b\, \texttt{then}\, y\, \texttt{else}\, z$" defines x by y if b is true and by z if b is false. It can be used without alternative "$x = \texttt{if}\, b\, \texttt{then}\, y$" to sample y at the clock b, as shown in figure 11.

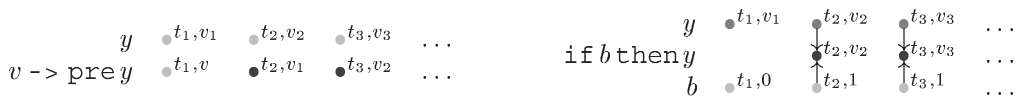

Figure 11: The if-then-else conditional in Lustre

Lustre programs are structured as data-flow functions, also called *nodes*. A node takes a number of input signals and defines a number of output signals upon the presence of an *activation* condition. If that condition matches an edge of the input signal clock, then the node is activated and possibly produces output. Otherwise, outputs are undetermined or defaulted. As an example, figure 12 defines a resettable counter. It takes an input signal `tick` and returns the `count` of its occurrences. A boolean `reset` signal can be triggered to reset the count to 0. We observe that the boolean input signals `tick` and `reset` are synchronous to the output signal `count` and define a data-flow function.

```
node counter (tick, reset: bool) returns (count: int);
let
    count = if   true->reset
            then 0
            else if tick then pre count+1 else pre count;
```

Figure 12: A resettable counter in Lustre

Combinators for Signal As opposed to nodes in Lustre, equations $x := y\ f\ z$ in Signal more generally denote processes that define timing relations between input and output signals. There are three primitive combinators in Signal:

- **Delay:** "$x := y\$1\ \texttt{init}\, v$" initially defines the signal x by the value v and then by the previous value of the signal y. The signal y and its delayed copy "$x := y\$1\ \texttt{init}\, v$" are synchronous: they share the same set of tags $t_1, t_2, \ldots$. Initially (at t_1), the signal x takes the declared value v. At tag t_n, x takes the value of y at tag t_{n-1}. This is displayed in figure 13.

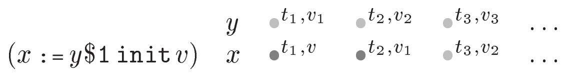

Figure 13: The delay operator in Signal

- **Sampling:** "$x := y\ \texttt{when}\ z$" defines x by y when z is true (and both y and z are present); x is present with the value v_2 at t_2 only if y is present with v_2 at t_2 and if z is present at t_2 with the value true. When this is the case, one needs to schedule the calculation of y and z before x, as depicted by $y_{t_2} \rightarrow x_{t_2} \leftarrow z_{t_2}$.
- **Merge:** "$x = y\ \texttt{default}\ z$" defines x by y when y is present and by z otherwise. If y is absent and z present with v_1 at t_1 then x holds (t_1, v_1). If y is present (at t_2 or t_3) then x holds its value whether z is present (at t_2) or not (at t_3). This is depicted in figure 14.

The structuring element of a Signal specification is a process. A process accepts input signals originating from possibly different clock domains to produce output signals when needed. Recalling the example of the resettable counter (figure 12), this allows, for instance, to specify a counter (pictured in figure 15) where the inputs `tick` and `reset` and the output `value` have independent clocks. The body of `counter` consists of one equation that defines

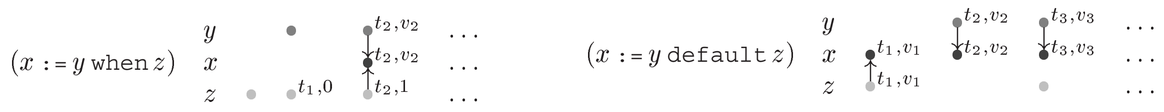

Figure 14: The merge operator in Signal

the output signal `value`. Upon the event `reset`, it sets the count to 0. Otherwise, upon a `tick` event, it increments the count by referring to the previous value of `value` and adding 1 to it. Otherwise, if the count is solicited in the context of the counter process (meaning that its clock is active), the counter just returns the previous count without having to obtain a value from the `tick` and `reset` signals.

```
process counter = (? event tick, reset ! integer value)
     (| value := (0 when reset)
         default ((value$ init 0 + 1) when tick)
         default (value$ init 0)
      |);
```

Figure 15: A resettable counter in Signal

A Signal process is a structuring element akin to a hierarchical block diagram. A process may structurally contain sub-processes. A process is a generic structuring element that can be specialized to the timing context of its call. For instance, a definition of the Lustre counter (figure 12) starting from the specification of figure 15 consists of the refinement depicted in figure 16. The input tick and reset clocks expected by the process `counter` are sampled from the boolean input signals `tick` and `reset` by using the "`when tick`" and "`when reset`"expressions. The count is then synchronized to the inputs by the equation `reset ^= tick ^= count`.

```
process synccounter = (? boolean tick, reset ! integer value)
     (| value := counter (when tick, when reset)
      | reset ^= tick ^= value
      |);
```

Figure 16: Synchronization of the counter interface

4.3 Compilation of declarative formalisms

The analysis and code generation techniques of Lustre and Signal are necessarily different, tailored to handle the specific challenges determined by the different models of computation and programming paradigms.

4.3.1 Compilation of Signal

Sequential code generation starting from a Signal specification starts with an analysis of its implicit synchronization and scheduling relations. This analysis yields the control and data flow graphs that define the class of sequentially executable specifications and allow to generate code.

Synchronization and scheduling analysis In SIGNAL, the clock $\hat{}x$ of a signal x denotes the set of instants at which the signal x is present. It is represented by a signal that is true when x is present and that is absent otherwise. Clock expressions (see figure 17) represent control. The clock "$\texttt{when}\, x$" (resp. "$\texttt{when not}\, x$") represents the time tags at which a boolean signal x is present and true (resp. false). The empty clock is denoted by 0. Clock expressions are obtained using conjunction, disjunction and symmetric difference over other clocks. Clock equations (also called clock relations) are Signal processes: the equation "$e \hat{}= e'$" synchronizes the clocks e and e' while "$e \hat{}< e'$" specifies the containment of e in e'. Explicit scheduling relations "$x \to y \,\texttt{when}\, e$" allow the representation of causality in the computation of signals (e.g. x after y at the clock e).

$$\begin{array}{rcll} e & ::= & \hat{}x \mid \texttt{when}\, x \mid \texttt{when not}\, x \mid e \,\hat{}\texttt{+}\, e' \mid e \,\hat{}\texttt{-}\, e' \mid e \,\hat{}\texttt{*}\, e' \mid 0 & \text{(clock expression)} \\ E & ::= & () \mid e \hat{}{=} e' \mid e \hat{}{<} e' \mid x \to y \,\texttt{when}\, e \mid E \,|\, E' \mid E/x & \text{(clock relations)} \end{array}$$

Figure 17: The syntax of clock expressions and clock relations (equations)

A system of clock relations E can be easily associated (using the inference system $P : E$ of figure 18) to any Signal process P, to represent its timing and scheduling structure.

$$\begin{array}{r} x \,\texttt{:=}\, y\$1 \,\texttt{init}\, v : \hat{}x \hat{}{=} \hat{}y \\ x \,\texttt{:=}\, y \,\texttt{when}\, z : \hat{}x \hat{}{=} \hat{}y \,\texttt{when}\, z \,|\, y \to x \,\texttt{when}\, z \\ x \,\texttt{:=}\, y \,\texttt{default}\, z : \hat{}x \hat{}{=} \hat{}y \,\hat{}\texttt{+}\, \hat{}z \,|\, y \to x \,|\, z \to x \,\texttt{when}\, (\hat{}z \,\hat{}\texttt{-}\, \hat{}y) \end{array} \qquad \frac{P : E \quad Q : E'}{P \,|\, Q : E \,|\, E'} \qquad \frac{P : E}{P/x : E/x}$$

Figure 18: The clock inference system of Signal

Hierarchization The clock and scheduling relations E of a process P define the control-flow and data-flow graphs that hold all necessary information to compile a Signal specification upon satisfaction of the property of *endochrony*, as illustrated in figure 19. A process is said endochronous *iff* given a set of input signals (x and y in figure 19) and flow-equivalent input behaviors (datagrams on the left of figure 19), it has the capability to reconstruct a unique synchronous behavior up to clock-equivalence: the datagrams of the input signals in the middle of figure 19 and of the output signal on the right of figure 19 are ordered in clock-equivalent ways.

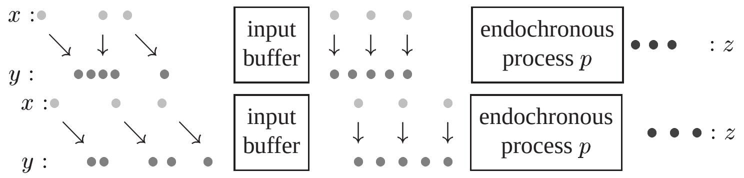

Figure 19: Endochrony: from flow-equivalent inputs to clock-equivalent outputs

To determine the order $x \preceq y$ in which signals are processed during the period of a reaction, clock relations E play an essential role. The process of determining this order is called hierarchization and consists of an insertion algorithm which proceeds in three easy steps:

1. First, equivalence classes are defined between signals of same clock: if $E \Rightarrow \hat{}x \hat{}{=} \hat{}y$ then $x \preceq y$ (we write $E \Rightarrow E'$ *iff* E implies E').
2. Second, elementary partial order relations are constructed between sampled signals: if $E \Rightarrow \hat{}x \hat{}{=} \texttt{when}\, y$ or $E \Rightarrow \hat{}x \hat{}{=} \texttt{when not}\, y$ then $y \preceq x$.
3. Last, assume a partial order of maximum z such that $E \Rightarrow \hat{}z = \hat{}y\, f\, \hat{}w$ (for some $f \in \{\, \hat{}\texttt{+}\,,\, \hat{}\texttt{*}\,,\, \hat{}\texttt{-}\,\}$) and a signal x such that $y \preceq x \succeq w$, then insertion consist of attaching z to x by $x \preceq z$.

The insertion algorithm proposed in [1] yields a canonical representation of the partial order $\preceq$ by observing that there exists a unique minimum clock x below z such that rule 3 holds. Based on the order $\preceq$, one can decide whether E is *hierarchical* by checking that its clock relation $\preceq$ has a minimum, written $\min_{\preceq} E \in \mathrm{vars}(E)$, so that $\forall x \in \mathrm{vars}(E), \exists y \in \mathrm{vars}(E), y \preceq x$. If E is furthermore *acyclic* (i.e. $E \Rightarrow x \to x \,\texttt{when}\, e$ implies $E \Rightarrow e \hat{}{=} 0$, for all $x \in \mathrm{vars}(E)$) then the analyzed process is endochronous, as shown in [33].

Example The implications of hierarchization for code generation can be outlined by considering the specification of one-place buffer in Signal (figure 20, left). Process `buffer` implements two functionalities. One is the process `alternate` which desynchronizes the signals `i` and `o` by synchronizing them to the true and false values of an alternating boolean signal `b`. The other functionality is the process `current`. It defines a `cell` in which values are stored at the input clock `^i` and loaded at the output clock `^o`. `cell` is a predefined Signal operation defined by:

$$x := y\ \mathtt{cell}\ z\ \mathtt{init}\ v =^{def} (m := x\$1\ \mathtt{init}\ v \,|\, x := y\ \mathtt{default}\ m \,|\, \hat{}x \hat{}{=}\ \hat{}y\ \hat{}{+}\ \hat{}z)\ /m$$

Clock inference (figure 20, middle) applies the clock inference system of figure 18 to the process `buffer` to determine three synchronization classes. We observe that `b, c_b, zb, zo` are synchronous and define the master clock synchronization class of `buffer`. There are two other synchronization classes, `c_i` and `c_o`, that corresponds to the true and false values of the boolean flip-flop variable `b`, respectively :

$$\mathtt{b} \leftrightarrow \mathtt{c\_b} \leftrightarrow \mathtt{zb} \leftrightarrow \mathtt{zo} \text{ and } \mathtt{b} \preceq \mathtt{c\_i} \leftrightarrow \mathtt{i} \text{ and } \mathtt{b} \preceq \mathtt{c\_o} \leftrightarrow \mathtt{o}$$

This defines three nodes in the control-flow graph of the generated code (figure 20, right). At the main clock `c_b`, `b` and `c_o` are calculated from `zb`. At the sub-clock `b`, the input signal `i` is read. At the sub-clock `c_o` the output signal `o` is written. Finally, `zb` is determined. Notice that the sequence of instructions follows the scheduling relations determined during clock inference.

```
process buffer = (? i ! o)
  (| alternate (i, o)
   | o := current (i)
   |) where
process alternate = (? i, o ! )
  (| zb := b$1 init true
   | b := not zb
   | o ^= when not b
   | i ^= when b
   |) / b, zb;
process current = (? i ! o)
  (| zo := i cell ^o init false
   | o  := zo when ^o
   |) / zo;
```

```

(| c_b ^= b
 | b   ^= zb
 | zb  ^= zo
 | c_i := when b
 | c_i ^= i
 | c_o := when not b
 | c_o ^= o
 | i -> zo when ^i
 | zb -> b
 | zo -> o when ^o
 |) / zb, zo, c_b,
       c_o, c_i, b;
```

```
buffer_iterate () {
  b = !zb;
  c_o = !b;
  if (b) {
    if (!r_buffer_i(&i))
      return FALSE;
  }
  if (c_o) {
    o = i;
    w_buffer_o(o);
  }
  zb = b;
  return TRUE;
}
```

Figure 20: Specification, clock analysis and code generation in Signal

4.3.2 Compilation of Lustre

Whereas Signal uses a hierarchization algorithm to find a sequential execution path starting from a system of clock relations, Lustre leaves this task to engineers, which must provide a sound, fully synchronized program in the first place: well-synchronized Lustre programs correspond to hierarchized Signal specifications.

The classic compilation of Lustre starts with a static program analysis that checks the correct synchronization and cycle freedom of signals defined within the program. Then, it essentially partitions the program into elementary blocks activated upon boolean conditions [35] and focuses on generating efficient code for high-level constructs, such as iterators for array processing [42].

Recent efforts have been conducted to enhance this compilation scheme by introducing effective activation clocks, whose soundness is checked by typing techniques. In particular, this was applied to the industrial SCADE version, with extensions [26, 25].

4.3.3 Certification

The simplicity of the single-clocked model of Lustre eases program analysis and code generation. Therefore, its commercial implementation – Scade by Esterel Technologies – provides a certified C code generator. Its combination to Sildex (the commercial implementation of Signal by TNI-Valiosys) as a front-end for architecture mapping and early requirement specification is the methodology advocated in the IST project Safeair (URL: `http://www.safeair.org`). The formal validation and certification of synchronous program properties has been the subject of numerous studies. In [44], a co-inductive axiomatization of Signal in the proof assistant Coq [31], based on the calculus of constructions [57], is proposed.

The application of this model is two-fold. It allows, first of all, for the exhaustive verification of formal properties of infinite-state systems. Two case studies have developed. In [36], a faithful model of the steam-boiler problem was given in Signal and its properties proved with Signal's Coq model. In [37], it is applied to proving the correctness of real-time properties of a protocol for loosely time-triggered architectures, extending previous work proving the correctness of its finite-state approximation [8].

Another important application of modeling Signal in the proof assistant Coq is being explored: the development of a reference compiler translating Signal programs into Coq assertions. This translation allows to represent model transformations performed by the Signal compiler as correctness-preserving transformations of Coq assertions, yielding a costly yet correct-by-construction synthesis of the target code.

Other approaches to the certification of generated code have been investigated. In [46], validation is achieved by checking a model of the C code generated by the Signal compiler in the theorem prover PVS with respect to a model of its source specification (translation validation).

Related work on modeling Lustre have equally been numerous and started in [45] with the verification of a sequential multiplier using a model of stream functions in Coq. In [21], the verification of Lustre programs is considered under the concept of generating proof obligations and by using PVS. In [19], a semantics of Lucid-Synchrone, an extension of Lustre with higher-order stream functions, is given in Coq.

5 Success stories – a viable approach for system design

Synchronous and reactive formalisms appeared in the early nineties and the theory matured and expanded since then to cover all the topics presented in this article. Research groups were active mostly in France, but also notably in Germany and in the US. Several large academic projects were completed, including the IST Syrf, Sacres and Safeair projects, as well as industrial early-adopters ones.

S/R modeling and programming environments are today marketed by two French software houses, Esterel Technologies for Esterel and SCADE/Lustre, and TNI-Valiosys for Sildex/Signal. The influence of S/R systems tentatively pervaded to hardware CAD products such as Synopsys CoCentric Studio and Cadence VCC, despite the omnipotence of classical HDLs there. The Ptolemy co-simulation environment from UC Berkeley comprises a S/R domain based on the synchronous hypothesis.

There have been a number of industrial take-ups on S/R formalisms, most of them in the aeronautics industry. Airbus Industries is now using Scade for the real design of parts of the new Airbus A-380 aircraft. S/R languages are also used by Dassault Aviation (for the next-generation Rafale fighter jet) and Snecma ([7] gives an in-depth coverage of these prominent collaborations). Car and phone manufacturers are also paying increasing attention (for instance at Texas Instruments), as well as advanced development teams in embedded hardware divisions of prominent companies (such as Intel).

6 Into the future: perspectives and extensions

Future advances in and around synchronous languages can be predicted in several directions:

Certified compilers. As already seen, this is the case for the basic SCADE compiler. But as the demand becomes higher, due to the critical-safety aspects of applications (in transportation fields notably), the impact of full-fledged operational semantics backing the actual compilers should increase.

Formal models and embedded code targets. Following the trend of exploiting formal models and semantic properties to help define efficient compilation and optimization techniques, one can consider the case of targeting distributed platforms (but still with a global reaction time). Then, the issues of spatial mapping and temporal scheduling of elementary operations composing the reaction inside a given interconnect topology become a fascinating (and NP-complete) problem. Heuristics for user guidance and semi-automatic approaches are the main topic of the SynDEx environment [39, 32]. Of course this requires an estimation of the time budgets for the elementary operations and communications.

Desynchronized systems. In larger designs, the full global synchronous assumption is hard to maintain, especially if long propagation chains occur inside a single reaction (in hardware, for instance, the clock tree cannot be distributed to the whole chip). Several types of answers are currently being brought to this issue, trying to instill a looser coupling of synchronous modules in a desynchronised network (one then talks of "Globally-Asynchronous Locally-Synchronous" systems). In the theory of *latency-insensitive design*, all processes are supposed to be able to stall until the full information is synchronously available. The exact latency duration meant to recover a (slower) synchronous model are computed afterwards, only after functional correctness on the more abstract level is achieved [22, 49]. Fancier approaches, trying to save on communications and synchronizations, are introduced in section 6.1.

Relations between transactional and cycle-accurate levels. If synchronous formalisms can be seen as a global attempt at transferring the notion of cycle-accurate modeling to the design of SW/HW embedded systems, then the existing gap between these levels must also be reconsidered in the light of formal semantics and mathematical models. Currently, there exists virtually no automation for the synthesis of RTL from TLM levels. The previous item, with its well-defined relaxation of synchronous hypothesis at specification time, could be a definite step in this direction (of formally linking two distinct levels of modeling).

Relations between cycle-accurate and timed models. Physical timing is of course a big concern in synchronous formalisms, if only to validate the synchronous hypothesis and establish converging stabilization of all values across the system before the next clock tick. While in traditional software implementations one can *decide* that the instant is over when all treatments were effectively completed, in hardware or other real-time distributed settings a true compile-time timing analysis is in order. Several attempts have been made in this direction [41, 24].

6.1 Asynchronous implementation of synchronous specifications

The relations between synchronous and asynchronous models have long remained unclear, but investigations in this direction have recently received an boost due to demands coming from the engineering world. The problem is that many classes of embedded applications are best modeled (at least in part) under the cycle-based synchronous paradigm, while their desired implementation is not. This problem covers implementation classes that become increasingly popular (such as distributed software or even complex digital circuits like the Systems-on-a-Chip), hence the practical importance of the problem. Such implementations are formed of components that are only loosely connected through communication lines that are best modeled as asynchronous. At the same time, the existing synchronous tools for specification, verification, and synthesis are very efficient and popular, meaning that they should be used for most of the design process.

In distributed software, the need for global synchronization mechanisms always existed. However, in order to be used in aerospace and automotive applications, an embedded system must also satisfy very high requirements in the areas of safety, availability, and fault tolerance. These needs prompted the development of integrated platforms, such as **TTA** [38], which offer higher-level, proven synchronization primitives, more adapted to specification, verification, and certification. The same correctness and safety goals are followed in a purely synchronous framework by two approaches: The AAA methodology and the **SynDEx** software of Sorel *et al.*[32] and the **Ocrep** tool of Girault *et al.*[23]. Both approaches take as input a synchronous specification, an architecture model, and some real-time and embedding constraints, and produce a distributed implementation that satisfies the constraints *and* the synchrony hypothesis (supplementary signals simulate at run-time the global clock of the initial specification). The difference is that Ocrep is rather tailored for control-dominated synchronous programs, while SynDEx works best on data-flow specifications with simple control.

In the (synchronous) hardware world, problems appear when the clock speed and circuit size become large enough to make global synchrony unfeasible (or at least very expensive), most notably in what concerns the distribution of the clock and the transmission of data over long wires between functional components. The problem is to insure that no communication error occurs due to the clock skew or due to the interconnect delay between the emitter and the receiver. Given the high cost (in area and power consumption) of precise clock distribution, it appears in fact that the only long-term solution is the division of large systems into several clocking domains, accompanied by the use of novel on-chip communication and synchronization techniques.

When the multiple clocks are strongly correlated, we talk about *mesochronous* or *plesiochronous* systems. However, when the different clocks are unrelated (*e.g.* for power saving reasons), the resulting circuit is best modeled as a *Globally Asynchronous Locally Synchronous (GALS)* system where the synchronous domains are connected through asynchronous communication lines (*e.g.* FIFOs). Such approaches are *pausible clocking* by Yun and Donohue [58], or, in a framework where a global, reference clock is still preserved, *latency-insensitive design* by Carloni and Sangiovanni-Vincentelli[22]. A multi-clock extension of the Esterel language[15] has been proposed for the description of such systems. A more radical approach to the hardware implementation of a synchronous specification is *desynchronization*[16], where the clock subsystem is entirely removed and replaced with asynchronous handshake logic. The advantages of such implementations are those of asynchronous logic: smaller power consumption, average-case performance, smaller electro-magnetic interference.

At an implementation-independent level, several approaches propose solutions to various aspects of the problem of GALS implementation. The *loosely time-triggered architectures*[8] of Benveniste *et al.* define a sampling-based approach to (inter-process) FIFO construction. More important, Benveniste *et al.*[5] define *semantics preservation* – an abstract notion of correct GALS implementation of a synchronous specification (asynchronous communication is modeled here as message passing). Latency insensitivity insures in a very simple, highly constrained way the semantics preservation. Less constraining and higher-level conditions are the compositional criteria of *finite flow-preservation* of Talpin *et al.* [54, 53] and of *weak endo-isochrony* of Potop, Caillaud, and Benveniste [48]. While finite-flow-preservation focuses on checking equivalence through finite desynchronization protocols, weak endo-isochrony allows to exploit the internal concurrency of synchronous systems in order to minimize signalization, and to handle infinite behaviors.

References

[1] Pascalin Amagbegnon, Loïc Besnard, and Paul Le Guernic. Implementation of the data-flow synchronous language Signal. In *Conference on Programming Language Design and Implementation (PLDI'95)*. ACM Press, 1995.

[2] Charles André. Representation and Analysis of Reactive Behavior: a Synchronous Approach. In *Computational Engineering in Systems Applications (CESA'96)*, pages 19–29. IEEE-SMC, 1996.

[3] Laurent Arditi, Hédi Boufaïed, Arnaud Cavanié, and Vincent Stehlé. Coverage-directed generation of system-level test cases for the validation of a DSP system. volume 2021 of *Lecture Notes in Computer Science*, 2001.

[4] Albert Benveniste and Gérard Berry. The synchronous approach to reactive and real-time systems. *Proceedings of the IEEE*, 79(9):1270–1282, September 1991.

[5] Albert Benveniste, Benoît Caillaud, and Paul Le Guernic. Compositionality in dataflow synchronous languages: Specification and distributed code generation. *Information and Computation*, 163:125–171, 2000.

[6] Albert Benveniste, Paul Caspi, Luca Carloni, and Alberto Sangiovanni-Vincentelli. Heterogeneous reactive systems modeling and correct-by-construction deployment. In *Embedded Software Conference (EMSOFT'03)*. Springer Verlag, October 2003.

[7] Albert Benveniste, Paul Caspi, Stephen Edwards, Nicolas Halbwachs, Paul Le Guernic, and Robert de Simone. Synchronous languages twelve years later. *Proceedings of the IEEE*, January 2003. Special issue on embedded systems.

[8] Albert Benveniste, Paul Caspi, Paul Le Guernic, Hervé Marchand, Jean-Pierre Talpin, and Stavros Tripakis. A protocol for loosely time-triggered architectures. In *Embedded Software Conference (EMSOFT'02)*, volume 2491 of *Lecture Notes in Computer Science*. Springer Verlag, October 2002.

[9] Albert Benveniste, Paul Le Guernic, and Christian Jacquemot. Synchronous programming with events and relations: the Signal language and its semantics. *Science of Computer Programming*, 16, 1991.

[10] Gérard Berry. Real-time programming: General-purpose or special-purpose languages. In G. Ritter, editor, *Information Processing 89*, pages 11–17. Elsevier Science Publishers B.V. (North Holland), 1989.

[11] Gérard Berry. Esterel on hardware. *Philosophical Transactions of the Royal Society of London, Series A*, 19(2):87–152, 1992.

[12] Gérard Berry. *The Constructive Semantics of Pure Esterel*. Esterel Technologies, electronic version available at http://www.esterel-technologies.com, 1999.

[13] Gérard Berry and Laurent Cosserat. The synchronous programming language Esterel and its mathematical semantics. volume 197 of *Lecture Notes in Computer Science*, 1984.

[14] Gérard Berry and Georges Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[15] Gérard Berry and Ellen Sentovich. Multiclock Esterel. In *Proceedings CHARME'01*, volume 2144 of *Lecture Notes in Computer Science*, 2001.

[16] Ivan Blunno, Jordi Cortadella, Alex Kondratyev, Luciano Lavagno, Kelvin Lwin, and Christos Sotiriou. Handshake protocols for de-synchronization. In *Proceedings of the International Symposium on Asynchronous Circuits and Systems (ASYNC'04)*, Crete, Greece, 2004.

[17] Amar Bouali. Xeve, an Esterel verification environment. In *Proceedings of the Tenth International Conference on Computer Aided Verification (CAV'98)*, volume 1427 of *Lecture Notes in Computer Science*, UBC, Vancouver, Canada, June 1998.

[18] Amar Bouali, Jean-Paul Marmorat, Robert de Simone, and Horia Toma. Verifying synchronous reactive systems programmed in es terel. In *Proceedings FTRTFT'96*, volume 1135 of *Lecture Notes in Computer Science*, pages 463–466, 1996.

[19] Sylvain Boulme and Grégoire Hamon. Certifying synchrony for free. In *Logic for Programming, Artificial Intelligence and Reasoning*, volume 2250 of *Lectures Notes in Artificial Intelligence*. Springer Verlag, 2001.

[20] Frédéric Boussinot and Robert de Simone. The Esterel language. *Proceedings of the IEEE*, September 1991.

[21] Cécile Dumas Canovas and Paul Caspi. A PVS proof obligation generator for Lustre programs. In *International Conference on Logic for Programming and Reasonning*, volume 1955 of *Lectures Notes in Artificial Intelligence*. Springer Verlag, 2000.

[22] Luca Carloni, Ken McMillan, and Alberto Sangiovanni-Vincentelli. The theory of latency-insensitive design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 20(9):1059–1076, 9 2001.

[23] Paul Caspi, Alain Girault, and Daniel Pilaud. Automatic distribution of reactive systems for asynchronous networks of processors. *IEEE Transactions on Software Engineering*, 25(3):416–427, 1999.

[24] Etienne Closse, Michel Poize, Jacques Pulou, Joseph Sifakis, Patrick Venier, Daniel Weil, and Sergio Yovine. TAXYS: a tool for the developpment and verification real-time embedded systems. In *Proceedings CAV'01*, volume 2102 of *Lecture Notes in Computer Science*, 2001.

[25] Jean-Louis Colaço, Alain Girault, Grégoire Hamon, and Marc Pouzet. Towards a higher-order synchronous data-flow language. In *Proceedings EMSOFT'04*, 2004.

[26] Jean-Louis Colaço and Marc Pouzet. Clocks as first class abstract types. In *Proceedings EMSOFT'03*, 2003.

[27] Robert de Simone and Annie Ressouche. Compositional semantics of Esterel and verification by compositional reductions. In *Proceedings CAV'94*, volume 818 of *Lecture Notes in Computer Science*, 1994.

[28] Stephen Edwards. An Esterel compiler for large control-dominated systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 21(2), February 2002.

[29] Robert French, Monica Lam, Jeremy Levitt, and Kunle Olukotun. A general method for compiling event-driven simulations. In *Proceedings of the 32nd Design Automation Conference (DAC'95)*, San Francisco, CA, USA, 1995.

[30] Minxi Gao, Jie-Hong Jiang, Yunjian Jiang, Yinghua Li, Subarna Sinha, and Robert Brayton. MVSIS. In *Proceedings of the International Workshop on Logic Synthesis (IWLS'01)*, Tahoe City, June 2001.

[31] Eduardo Giménez. *Un Calcul de Constructions Infinies et son Application à la Vérification des Systèmes Communicants*. PhD thesis, Laboratoire de l'Informatique du Parallélisme, Ecole Normale Supérieure de Lyon, December 1996.

[32] Thierry Grandpierre, Christophe Lavarenne, and Yves Sorel. Optimized rapid prototyping for real time embedded heterogeneous multiprocessors. In *Proceedings of the 7th International Workshop on Hardware/Software Co-Design (CODES'99)*, Rome, 1999.

[33] Paul Le Guernic, Jean-Pierre Talpin, and Jean-Christophe Le Lann. Polychrony for system design. *Journal of Circuits, Systems and Computers. Special Issue on Application-Specific Hardware Design*, 2002.

[34] Nicolas Halbwachs. Synchronous programming of reactive systems. In *Computer Aided Verification (CAV'98)*, pages 1–16, 1998.

[35] Nicolas Halbwachs, Paul Caspi, and Pascal Raymond. The synchronous data-flow programming language Lustre. *Proceedings of the IEEE*, 79(9), 1991.

[36] Mickael Kerboeuf, David Nowak, and Jean-Pierre Talpin. Specification and verification of a steam-boiler with Signal-Coq. In *International Conference on Theorem Proving in Higher-Order Logics*, volume 1869 of *Lectures Notes in Computer Science*. Springer Verlag, 2000.

[37] Mickael Kerboeuf, David Nowak, and Jean-Pierre Talpin. Formal proof of a polychronous protocol for loosely time-triggered architectures. In *International Conference on Formal Engineering Methods*, volume 2885 of *Lectures Notes in Computer Science*. Springer Verlag, 2003.

[38] Hermann Kopetz. *Real-Time Systems, Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, 1997.

[39] Christophe Lavarenne, Omar Seghrouchni, Yves Sorel, and Michel Sorine. The SynDEx software environment for real-time distributed systems design and implementation. In *ECC'91*, 1991.

[40] Jaejin Lee, David Padua, and Samuel Midkiff. Basic compiler algorithms for parallel programs. In *Proceedings of the 7th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Atlanta, GA, USA, 1999.

[41] George Logothetis and Klaus Schneider. Exact high-level WCET analysis of synchronous programs by symbolic state space exploration. In *Proceedings DATE2003*, 2003.

[42] Florence Maraninchi and Lionel Morel. Arrays and contracts for the specification and analysis of regular systems. In *International Conference on Applications of Concurrency to System Design (ACSD'04)*. IEEE Press, 2004.

[43] Steven Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, 1997.

[44] David Nowak, Jean-Rene Beauvais, and Jean-Pierre Talpin. Co-inductive axiomatization of a synchronous language. In *International Conference on Theorem Proving in Higher-Order Logics*, volume 1479 of *Lectures Notes in Computer Science*. Springer Verlag, 1998.

[45] Christine Paulin-Mohring. Circuits as streams in Coq : Verification of a sequential multiplier. In S. Berardi and M. Coppo, editors, *Types for Proofs and Programs, TYPES'95*, volume 1158 of *Lecture Notes in Computer Science*, 1996.

[46] Amir Pnueli, O. Shtrichman, and M. Siegel. Translation validation: from Signal to C. In *Correct System Design Recent Insights and Advance*, volume 1710 of *Lecture Notes in Computer Science*. Springer Verlag, 2000.

[47] Dumitru Potop-Butucaru and Robert de Simone. Optimizations for faster execution of Esterel programs. In Rajesh Gupta, Paul Le Guernic, Sandeep Shukla, and Jean-Pierre Talpin, editors, *Formal Methods and Models for System Design*, 2004. Kluwer.

[48] Dumitru Potop-Butucaru, Benoît Caillaud, and Albert Benveniste. Concurrency in synchronous systems. In *International conference on applications of concurrency to system design (ACSD'04)*. IEEE Press, 2004.

[49] Alberto Sangiovanni-Vincentelli, Luca Carloni, Fernando De Bernardinis, and Marco Sgroi. Benefits and challenges of platform-based design. In *Proceedings of the Design Automation Conference (DAC'04)*, 2004.

[50] Ellen Sentovich, Kanwar Jit Singh, Luciano Lavagno, Cho Moon, Rajeev Murgai, Alexander Saldanha, Hamid Savoj, Paul Stephan, Robert Brayton, and Alberto Sagiovanni-Vincentelli. SIS: A system for sequential circuit synthesis. Memorandum UCB/ERL M92/41, UCB, ERL, 1992.

[51] Ellen Sentovich, Horia Toma, and Gérard Berry. Latch optimization in circuits generated from high-level descriptions. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD'96)*, 1996.

[52] Tom Shiple, Gérard Berry, and Hervé Touati. Constructive analysis of cyclic circuits. In *Proceedings of the International Design and Testing Conference (ITDC)*, Paris, 1996.

[53] Jean-Pierre Talpin and Paul Le Guernic. Algebraic theory for behavioral type inference. *Formal Methods and Models for System Design, chapter VIII*, Kluwer Academic Press, 2004.

[54] Jean-Pierre Talpin, Paul Le Guernic, Sandeep Kumar Shukla, Frédéric Doucet, and Rajesh Gupta. Formal refinement checking in a system-level design methodology. *Fundamenta Informaticae*, IOS Press, 2004.

[55] Hervé Touati and Gérard Berry. Optimized controller synthesis using Esterel. In *Proceedings of the International Workshop on Logic Synthesis (IWLS'93)*, Lake Tahoe, 1993.

[56] Daniel Weil, Valérie Bertin, Etienne Closse, Michel Poize, Patrick Vernier, and Jacques Pulou. Efficient compilation of Esterel for real-time embedded systems. In *Proceedings CASES'00*, San Jose, CA, USA, 2000.

[57] Benjamin Werner. *Une Théorie des Constructions Inductives*. PhD thesis, Université Paris VII, Mai. 1994.

[58] Kenneth Yun and Ryan Donohue. Pausible clocking: A first step toward heterogenous systems. In *International Conference on Computer Design (ICCD'96)*, 1996.