Modular Compilation of a Synchronous Language

Annie Ressouche, Daniel Gaffé, Valérie Roy

Modular Compilation of a Synchronous Language

Annie Ressouche and Daniel Gaffé and Valérie Roy

Abstract Synchronous languages rely on formal methods to ease the development of applications in an efficient and reusable way. Formal methods have been advocated as a means of increasing the reliability of systems, especially those which are safety or business critical. It is still difficult to develop automatic specification and verification tools due to limitations like state explosion, undecidability, etc... In this work, we design a new specification model based on a reactive synchronous approach. Then, we benefit from a formal framework well suited to perform compilation and formal validation of systems. In practice, we design and implement a special purpose language (LE) and its two semantics : the *behavioral semantics* helps us to define a program by the set of its behaviors and avoid ambiguousness in programs' interpretation; the *execution equational semantics* allows the modular compilation of programs into software and hardware targets (C code, Vhdl code, Fpga synthesis, Verification tools). Our approach is pertinent considering the two main requirements of critical realistic applications : the modular compilation allows us to deal with large systems, the model-driven approach provides us with formal validation.

keywords:
model-driven language, synchronous models, compilation, modularity, verification

Annie Ressouche
INRIA Sophia Antipolis - Méditerranée 2004 route des Lucioles BP 93 06902 Sophia Antipolis, FRANCE, e-mail: Annie.Ressouche@sophia.inria.fr

Daniel Gaffe
LEAT Laboratory, Univ of Nice Sophia Antipolis CNRS 250 rue Albert Einstein 06560 Valbonne France, e-mail: daniel.gaffe@unice.fr

Valérie Roy
CMA Ecole des Mines Sophia Antipolis France, e-mail: vr@cma.ensmp.fr

1 Introduction

Synchronous languages [3, 1, 8] have been designed to specify reactive systems [10]. All are model-driven languages to allow both efficiency and reusability of system design, and formal verification of system behavior. They rely on the *synchronous hypothesis* which assumes a discrete logic time scale, made of instants corresponding to reactions of the system. All the events concerned by a reaction are simultaneous : input events as well as the triggered output events. As a consequence, a reaction is instantaneous (we consider that a reaction takes no time, in compliance with synchronous language class), there are no concurrent partial reactions and so determinism can be ensured.

Although synchronous languages have begun to face the state explosion problem, there is still a need for further research on efficient and modular compilation of synchronous languages. The first compilers translated the program into an extended finite state machine. The drawback of this approach remains the potential state explosion problem. Polynomial compilation was first achieved by a translation to equation systems that symbolically encode the automata. This approach is the core of commercial tool [17]. Then several approaches translate the program into event graphs [19] or concurrent data flow graphs [6, 14] to generate efficient C code. All these methods have been used to optimize the compilation times as well as the size and the execution of the generated code.

However none of these approaches consider a modular compilation. Of course there is a fundamental contradiction in relying on a formal semantics to compile reactive systems because a perfect semantics would combine three important properties: *responsiveness*, *modularity* and *causality*. Responsiveness means that we can deal with a logical time and we can consider that output events occur in the same reaction that the input events causing them. It is one of the foundations of the synchronous hypothesis. Causality means that for each event generated in a reaction, there is a causal chain of events leading to this generation. No causal loop may occur. A semantics is modular when “environment to component” and “component to component” communication is treated symmetrically [11]. In particular, the semantics of the composition of two reactive systems can be deduced from the respective semantics of each sub-part. Another aspect of modularity is the coherent view each subsystem has of what is going on. When an event is present, it is broadcasted all around the system and is immediately available for every part which listen to it. Unfortunately, there exists a theorem (“the RMC barrier theorem”) [11] that states that these three properties cannot be united in a semantics. Synchronous semantics are responsive and modular. But causality remains a problem in these semantics and modular compilation must be completed by a global causality checking.

In this paper we introduce a reactive synchronous language, we define its behavioral semantics that gives a meaning to programs and an equational semantics allowing first a modular compilation of programs and second an automatic verification of properties. As other synchronous semantics, we get a causality problem and we face it with the introduction of a new sorting algorithm that allows us to start from compiled subsystems to compile the overall system without sort again all the equations.

The paper is organized as follows: section 2 is dedicated to LE language: Its syntax is briefly described and its behavioral and equational semantics are both discussed. Section 3 details how we perform a separated compilation of LE programs. Then, we compare our approach with others in section 4. Finally, we conclude and open up the way for future works in section 5.

2 LE Language

2.1 Language Overview

LE language belongs to the family of reactive synchronous languages. It is a discrete control dominated language. Nevertheless, we benefit from a great many studies about synchronous language domain since two decades. As a consequence, we choose to not introduce the powerful trap exit mechanism existing in the Esterel synchronous language [3] since it is responsible for a large part of the complexity of compilation. We just keep an abortion operator that only allows to exit one block. It is not a strong restriction since we can mimic trap exit with cascade of abortions. On the other hand, our language offers an *automaton description* as a native construction. Moreover, our graphical tool (GALAXY) helps the user editing automata and generating the LE code.

nothing	does nothing
emit speed	signal *speed* is immediately present in the environment
present S { P1} else { P2}	If signal S is present $P1$ is performed otherwise $P2$
$P_1 \gg P_2$	perform P_1 then P_2
$P_1 \| P_2$	synchronous parallel: start P_1 and P_2 simultaneously and stop when both have terminated
$abort\ P\ when\ S$	perform P until an instant in which S is present
$loop\ \{P\}$	perform P and restart when it terminates
$local\ S\ \{P\}$	encapsulation, the scope of S is restricted to P
$Run\ M$	call of module M
$pause$	stop until the next reaction
$waitS$	stop until the next reaction in which S is present
$\mathcal{A}(\mathcal{M}, \mathcal{T}, \mathcal{C}ond, M_f, \mathcal{O}, \lambda)$	automata specification

Table 1 LE Operators

More precisely, LE language unit is a *named module*. The *module interface* declares the set of *input events* it reacts to and the set of *output events* it emits. In addition, the location of external already compiled sub modules is also specified in module interface. The *module body* is expressed using a set of *operators*. The language's operators and constructions are chosen to fit the description of reactive applications as a set of concurrent communicating sub-systems. Communication takes place between modules or between a module and its environment. Subsystems communicate

via *events*. Besides, some operators (wait, pause) are specially devoted to deal with the logical time. We do not detail LE operators, we define them in table 1 and a complete description can be found in [15]. But, we just underline the presence of the *run module* operator that calls an external module and supports a renaming of the interface signals of the called module. It is through this operator usage that modular compilation is performed.

2.2 LE Semantics

Now, we discuss LE semantics. Our approach is twofold: first, we define a *behavioral* semantics providing a consistent meaning to each LE program. Such a semantics defines a program by the set of its behaviors. Second, we propose an *equational* semantics to express programs as a set of equations and get a compilation means. We introduce these two semantics because we need both a well suited framework to apply verification techniques and an operational framework to get an effective compilation of programs.

2.2.1 Mathematical Context

Similarly to others synchronous reactive languages, LE handles *broadcasted signals* as communicating means. A program reacts to input events in producing output events. An *event* is a signal carrying some information related to its *status*. To get an easier way to check causality, we introduce a boolean algebra (called ξ) that provides us with a smarter information about event. ($\xi = \{\bot, 0, 1, \top\}$). Let S be a signal, S^x denotes its status in a reaction. More precisely, S^1 means that S is present, S^0 means that S is absent, $S^\bot$ means that S status has not been established (neither from the external context nor from internal information propagation), and finally $S^\top$ corresponds to an event the status of which cannot be induced because it has two incompatible status in two different sub parts of the program. For instance, if S is both absent and present, then it turns out to have $\top$ status and thus an error occurs. Indeed the set ξ is a complete lattice [1].
We define three internal composition laws in ξ: $\sqcup$, $\sqcap$ and $\neg$. The $\sqcup$ law yields the upper bound of its two operands. The $\sqcap$ law yields the lower bound of its two operands, while the $\neg$ law is an inverse law. The set ξ with these 3 operations verify the axioms of *Boolean Algebra*. As a consequence, we can apply classical results concerning Boolean algebras to solve equation systems whose variables belong to ξ. The equational semantics described in section 2.2.3 relies on boolean algebra properties to compute signal status as solution of ξ equations.
An *environment* E is a set of events built from an enumerable set of signals, where each signal has a single status (i.e if S^x and $S^y \in E$ then $x = y$). Environments are useful to record the current status of signals in a program reaction. We easily extend

[1] with respect to the order($\leq$): $\bot \leq 0 \leq \top$; $\bot \leq 1 \leq \top$; $\bot \leq \top$.

the operation defined in ξ to environments [2]. We define relation ($\preceq$) on environments as follows:

$$E \preceq E' \text{ iff } \forall S^x \in E, \exists S^y \in E' | S^x \leq S^y$$

Thus $E \preceq E'$ means that each element of E is less than an element of E' according to the lattice order of ξ. As a consequence,the set of environments built from a signal set $\mathcal{S}$ ordered with the $\preceq$ relation is a lattice and $\sqcup$ and $\sqcap$ operations are monotonic with respect to $\preceq$.

2.2.2 LE Behavioral Semantics

To define the behavioral semantics of LE programs, we formalize the notion of concurrent computation and we rely on an algebraic theory that allows the description of behaviors in a formal way. The behavioral semantics formalizes a reaction of a program P according to an event input set. $(P, E) \longmapsto (P', E')$ has the usual meaning: E and E' are respectively input and output environments; program P reacts to E, reaches a new state represented by P' and the output environment is E'. This semantics supports a rule-based specification to describe the behavior of LE statement. A rule of the semantics has the form: $p \xrightarrow[E]{E', TERM} p'$ where p and p' are LE statements. E is an environment that specifies the status of the signals declared in the scope of p, E' is the output environment and $TERM$ is a boolean termination flag. Let P be a LE program and E an input event set, a reaction is computed as follows:

$$(P, E) \longmapsto (P', E') \quad \text{iff} \quad \Gamma(P) \xrightarrow[E]{E',\, TERM} \Gamma(P')$$

where $\Gamma(P)$ is the LE statement body of program P. We cannot detail all semantics definition due to a lack of space. To give a flavor of semantics rules, we show the rule for parallel operator. This operator respects the synchronous paradigm. It computes its two operands according to the broadcast of signals between each side and it is terminated when both sides are.

$$\frac{p \xrightarrow[E]{E_p,\, TERM_p} p' \quad , \quad q \xrightarrow[E]{E_q,\, TERM_q} q'}{p \| q \xrightarrow[E]{E_p \sqcup E_q,\, TERM_p . TERM_q} p' \| q'} \quad (parallel)$$

The behavioral semantics is a "macro" semantics that gives the meaning of a reaction for each LE statement. Nevertheless, a reaction is the least fixed point of a micro step semantics that computes the output environment from the initial one. At

[2] Let E and E' be 2 environments:

$$\begin{aligned} E \sqcup E' &= \{S^z | \exists S^x \in E, S^y \in E', z = x \sqcup y\} \\ &\quad \cup \{S^z | S^z \in E, \nexists S^y \in E'\} \\ &\quad \cup \{S^z | S^z \in E', \nexists S^y \in E\} \\ E \sqcap E' &= \{S^z | \exists S^x \in E, S^y \in E', z = x \sqcap y\} \\ \neg E &= \{S^x | \exists S^{\neg x} \in E\} \end{aligned}$$

each micro step, the environment is increased using the $\sqcup$ operation. According to the monotonicity of $\sqcup$ with respect to the $\preceq$ order and to the lattice character of the environment sets based on a given signal set, we can ensure that for each term, this least fixed point exists.

2.2.3 LE Equational Semantics

The behavioral semantics describes how the program reacts in an instant. It is logically correct in the sense that it computes a single output environment for each input event environment when there is no causality cycles. To face this inherent causality cycle problem specific to synchronous approach, constructive semantics have been introduced [2]. Such a semantics for synchronous languages is the application of constructive boolean logic theory to synchronous language semantics definition. The idea of constructive semantics is to "forbid self-justification and any kind of speculative reasoning replacing them by a fact-to-fact propagation"[2]. A program is *constructive* if and only if fact propagation is sufficient to establish the presence or absence of all signals. An elegant means to define a constructive semantics for a language is to translate each program into a constructive circuit. Such a translation ensures that programs containing no cyclic instantaneous signal dependencies are translated into cycle free circuits.
LE equational semantics respects this constructive principle. It computes output environments from input ones according to the ξ sequential circuits we associate with LE programs. Environments are built from the disjoint union of (1) a set of input, output and local signals, (2) a set of wires $\mathcal{W}$ and (3) a set of registers R (initially valued to 0). Registers are memories that feed back the circuit. The translation of a program into a circuit is structurally done. Sub-circuits are associated with program sub-terms. To this aim, we associate a circuit with each LE statement. Given p a LE statement, we call $\mathcal{C}(p)$ its associated ξ circuit. Each circuit $\mathcal{C}(p)$ has three interface wires: Set_p input wire activates the circuit, Reset_p input wire deactivates it and RTL_p is the "ready to leave" output wire. This latter indicates that the statement can terminate in the current clock instant. Reset_p wire is useful to deactivate a sub-circuit and top down propagate this deactivation. For instance, consider the sequence operator $(P_1 \gg P_2)$. Its circuit $\mathcal{C}(P_1 \gg P_2)$ is composed of $\mathcal{C}(P_1)$, $\mathcal{C}(P_2)$ and some equations defining its interface wires. The $\mathrm{Set}_{P_1 \gg P_2}$ input wire forwards down the control ($\mathrm{Set}_{P_1} = \mathrm{Set}_{P_1 \gg P_2}$). The $\mathrm{Reset}_{P_1 \gg P_2}$ input wire on one hand deactivates $\mathcal{C}(P_1)$ when its computation is over, and on the other hand, forwards down the deactivation information ($\mathrm{Reset}_{P_1} = \mathrm{Reset}_{P_1 \gg P_2} \sqcup \mathrm{RTL}_{P_1}$). Finally, the overall circuit is ready to leave when P_2 is ($\mathrm{RTL}_{P_1 \gg P_2} = \mathrm{RTL}_{P_2}$). Moreover, a circuit can need a register (denoted R) when the value of a wire or a signal is required for the next step computation.
The computation of circuit output environments is done according to a propagation law and to ensure that this propagation leads to logically correct solutions, a constructive value propagation law is supported by the computation and solutions of equation systems allow us to determine all signal status .

$$E \vdash bb \hookrightarrow bb \qquad \frac{E(w) = bb}{E \vdash w \hookrightarrow bb} \qquad \frac{E \vdash e \hookrightarrow bb}{E \vdash (w = e) \hookrightarrow bb} \qquad \frac{E \vdash e \hookrightarrow \neg bb}{E \vdash \neg e \hookrightarrow bb}$$

$$\frac{E \vdash e \hookrightarrow \top \text{ or } E \vdash e' \hookrightarrow \top}{E \vdash e \sqcup e' \hookrightarrow \top} \qquad \frac{E \vdash e \hookrightarrow \bot \text{ or } E \vdash e' \hookrightarrow \bot}{E \vdash e \sqcap e' \hookrightarrow \bot}$$

$$\frac{E \vdash e \hookrightarrow 1[0] \text{ and } E \vdash e' \hookrightarrow 0[1]}{E \vdash e \sqcup e' \hookrightarrow \top \text{ and } E \vdash e \sqcap e' \hookrightarrow \bot} \qquad \frac{E \vdash e \hookrightarrow 1[\bot] \text{ and } E \vdash e' \hookrightarrow \bot[1]}{E \vdash e \sqcup e' \hookrightarrow 1 \textit{ and } E \vdash e \sqcap e' \hookrightarrow \bot}$$

$$\frac{E \vdash e \hookrightarrow 0[\bot] \text{ and } E \vdash e' \hookrightarrow \bot[0]}{E \vdash e \sqcup e' \hookrightarrow 0} \qquad \frac{E \vdash e \hookrightarrow 0[\top] \text{ and } E \vdash e' \hookrightarrow \top[0]}{E \vdash e \sqcap e' \hookrightarrow 0}$$

$$\frac{E \vdash e \hookrightarrow x \text{ and } E \vdash e' \hookrightarrow x (x = \bot, 0, 1, \top)}{E \vdash e \sqcup e' \hookrightarrow x \text{ and } E \vdash e \sqcap e' \hookrightarrow x} \qquad \frac{E \vdash e \hookrightarrow 1[T] \text{ and } E \vdash e' \hookrightarrow T[1]}{E \vdash e \sqcap e' \hookrightarrow 1}$$

Table 2 Definition of the constructive propagation law ($\hookrightarrow$). The value between square brackets ([x]) means "respectively". For instance: $E \vdash e \hookrightarrow 1[0]$ and $E \vdash e' \hookrightarrow 0[1]$ means $E \vdash e \hookrightarrow 1$ and $E \vdash e' \hookrightarrow 0$ or $E \vdash e \hookrightarrow 0$ and $E \vdash e' \hookrightarrow 1$. $E(w)$ denotes the value of w in E.

Let $\mathcal{C}$ be a circuit, E an input environment, the constructive propagation law ($\hookrightarrow$) has the form : $E \vdash w \hookrightarrow bb$, where w is a ξ expression and bb is a ξ value; the law means that from E assignment values to signals and registers, w evaluates to bb. The $\hookrightarrow$ law is defined in table 2. By extension, let $\mathcal{C}$ be a circuit, $E \vdash \mathcal{C} \hookrightarrow E'$ means that E' is the result of the application of the propagation law to each equations of the circuit.

Now, we define the equational semantics $\mathcal{S}_e$. It is a mapping that computes an output environment from an input one and a LE statement. Let p be a LE statement, and E an environment, $\mathcal{S}_e(p, E) = \langle p \rangle_E$ iff $E \vdash \mathcal{C}(p) \hookrightarrow \langle p \rangle_E$. Finally, Let P be a LE program, the equational semantics also formalizes a reaction of a program P according to an input environment:$(P, E) \longmapsto E'$ iff $\mathcal{S}_e(\Gamma(P), E) = E'$.

To illustrate how the output environment is built and similarly to section 2.2.2, we focus on the parallel LE operator. As already said, each circuit $\mathcal{C}(p)$ has three specific connexion wires belonging to $\mathcal{W}$ (Set_p, Reset_p and RTL_p). For the parallel operator $(P_1 \| P_2)$, the output environment $\langle P_1 \| P_2 \rangle_E$ is the upper bound of the respective output environments of P_1 and P_2. The parallel is ready to leave when both P_1 and P_2 are. The rule for $\|$ is:

$\langle P_1 \rangle_E \sqcup \langle P_2 \rangle_E \vdash \mathcal{C}(P_1) \cup \mathcal{C}(P_2) \cup \mathcal{C}_{P_1 \| P_2} \hookrightarrow \langle P_1 \| P_2 \rangle_E$ where

$$\mathcal{C}_{P_1 \| P_2} = \begin{array}{ll} \mathrm{Set}_{P_1} = & \mathrm{Set}_{P_1 \| P_2} \\ \mathrm{Set}_{P_2} = & \mathrm{Set}_{\mathrm{P_1} \| \mathrm{P_2}} \\ \mathrm{Reset}_{P_1} = & \mathrm{Reset}_{P_1 \| P_2} \\ \mathrm{Reset}_{P_2} = & \mathrm{Reset}_{P_1 \| P_2} \\ \mathrm{R_1}^+ = & (\mathrm{RTL}_{P_1} \sqcup \mathrm{R_1}) \sqcap \neg \mathrm{Reset}_{\mathrm{P_1} \| \mathrm{P_2}} \\ \mathrm{R_2}^+ = & \mathrm{RTL}_{P_2} \sqcup \mathrm{R_2}) \sqcap \neg \mathrm{Reset}_{P_1 \| P_2} \\ \mathrm{RTL}_{P_1 \| P_2} = & (\mathrm{RTL}_{P_1} \sqcup \mathrm{R_1}) \sqcap (\mathrm{RTL}_{P_2} \sqcup \mathrm{R_2}) \end{array}$$

Notice that the circuit associated with the parallel operator requires two registers R_1 and R_2 to record the respective RTL values of both sides, because the parallel operator is ready to leave when its both arguments are.
The equational semantics provide us with an operational means to compile LE programs. It associates a ξ-equation system to each LE term and the evaluation of this last is done with respect to a constructive propagation law. A program is causal when we can sort its equation system. The equational semantics is responsive and modular. Then, this semantics is not causal. Although, we can rely on it to perform separated compilation of programs, We define a new sorting algorithm (based on the well-known `PERT` [18] technique) allowing to sort an equation system from already sorted sub parts. To complete our approach, we have shown that the two semantics presented in this paper coincide. For each reaction, given an input environment, if the program is causal, the equation system computes some output values and then the behavioral semantics computes identical outputs.

Theorem 1 *Let P be a* LE *statement, O its output signal set and $E_{\mathcal{C}}$ an input environment, the following property holds:*
$\Gamma(P) \xrightarrow[E]{E', \mathrm{RTL}(P)} \Gamma(P')$ *and* $\langle P \rangle_{E_{\mathcal{C}}} \restriction_O = E' \restriction_O$

where $E \restriction_X$ means the restriction of environment to the the events associated with signals in X. $E = \{S^x | S^x \in E_{\mathcal{C}} \; and \; S \notin \mathcal{W} \cup \mathcal{R}\}$. RTL_P (RTL wire of $\mathcal{C}(P)$) can be considered as a boolean because it cannot be evaluated to $\bot$ or $\top$ and its value is the termination flag of behavioral semantics.

3 LE Modular Compilation

3.1 Introduction

In the previous section, we have shown that every construct of the language has a semantics expressed as a set of ξ equations. The first compilation step is the generation of a ξ equation system for each LE program, according to the semantics laws described in section 2.2.3.
But to rely on equation systems to generate code, simulate or link with external code requires to find an evaluation order, valid for all synchronous instants. Usually, in the most popular synchronous languages existing, this order is static. This static order forbids any separated compilation mechanism as shown in the example in figure 1.

```
module first:
Input: I1,I2;
Output: O1,O2;
loop {
 pause >>
 {
 present I1 {emit O1}
 ||
 present I2 {emit O2}
 }
end
```

```
module second:
Input: I3;
Output: O3;
loop {
 pause >> present I3 {emit O3}
}
end
```

```
module final:
Input: I;
Output O;
local L1,L2 {
 run first[ L2\I1,O\O1,I\I2,L1\O2]
 ||
 run second[ L1\I3,L2\O3]
}
end
```

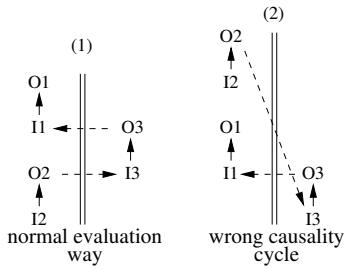
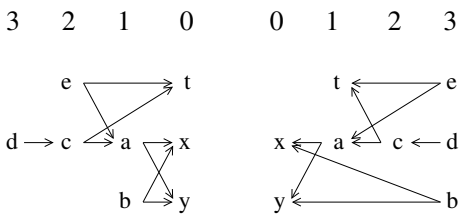


Fig. 1 Causality cycle generation. The `pause` instruction waits an instant to avoid instantaneous loop in modules **first** and **second**. In this example, O1, O2 and O3 signals are independent. Module first equation system is {O1 = I1, O2 = I2} and module second has { O3 = I3 } as equation system. But, when choosing a total order, we can introduce a causality cycle. If ordering (1) is chosen, in module final, taking into account the renaming, we obtain the system: { L1 = I, L2 = L1, O = L2 } which is well sorted. At the opposite, if we choose ordering (2), in module final we get: { L2 = L1, O = L2,L1 = I } which has a causality cycle.

To avoid such a drawback, independent signals must stay not related : we aim at building an incremental partial order. Hence, we keep enough information on signal causality to preserve the independence of signals. At this aim, we define two integer variables for each equation, namely (*CanDate*, *MustDate*) to record the date when the equation is evaluated. The CanDate characterizes the earliest date when the equation *can* be evaluated. The MustDate characterizes the latest date the equation *must* be evaluated to respect the critical time path of the equation system. Dates are ranges in a discrete time scale and we call them *levels*. Level 0 characterizes the equations evaluated first because they depend of free variables, while level n+1 characterizes the equations that require the evaluation of variables from lower levels (from n to 0) to be evaluated. Equations of same level are independent and so can be evaluated whatever the chosen order is. In fact, a couple of levels is associated with each equations. This methodology is inspired from the `PERT` method [18]. This latter is well known for decades in the industrial production. This technique allows to deal with partial orders in equation systems and not immediately force the choice of a total arbitrary order.

$a = x \sqcup y$
$b = x \sqcup not\ y$
$c = a \sqcup t$
$d = a \sqcup c$
$e = a \sqcup t$



a	b	c	d	e	x	y	t	
(1,1)	(1,3)	(2,2)	(3,3)	(2,3)	(0,0)	(0,0)	(0,2)	`variables CanDate and MustDate`

Table 3 An equation system and its upstream and downstream dependency graphs. the CanDate and MustDate table denotes the computed dates of each variable, for instance a has CanDate 1 and MustDate 1).

3.2 Sort algorithm: a PERT family method

Our sorting algorithm is divided into two phases. The first step constructs a forest where each tree represents variable dependencies. Thus initial partial orders sets are built. The second step is the recursive propagation of *CanDate* and *MustDate*. If during the propagation, a cycle is found there is a causality cycle in the program. Of course the propagation ends since the number of variables is finite. At worst, if the algorithm is successful (no causality cycle is found), we can find a total order with a single variable per level (n equations and n levels).

3.2.1 Sorting algorithm Description

More precisely, the first step builds two dependency sets (*upstream, downstream*) for each variable with respect to its associated equation. The *upstream* set of a variable x is composed of the variables needed by x to be computed while the *downstream* set is the variables that need the value of x to be evaluated. The toy example described in table 3 illustrates how we build upstream and downstream partial orders sets. First, from equation system, we build two dependency graphs. Each graph leads to construct two partial orders sets. For instance, in table 3, the upstream partial order set contains: $\{d \rightarrow c \rightarrow a \rightarrow x\}$, $\{d \rightarrow c \rightarrow a \rightarrow y\}$, $\{b \rightarrow x\}$, $\{b \rightarrow y\}$, etc....
After the construction of variable dependencies, we perform *CanDate* and *MustDate* propagation. Initially, all variables are considered independent and their dates (*CanDate , MustDate*) are set to (0,0). The second step recursively propagates the *CanDate* according to the upstream partial orders set while the *MustDate* is propagated according to the downstream partial orders set. In table 3, the CanDate and MustDate of each variable is shown. As a result, we have several orders acceptable. Looking at the example (see table 3), there are four levels (from 0 to 3). At level 0, we have x, y which are free variables. Equations for c is at level 2 and equation for

d is at level 3 while equation for b can be at each level between 1 and 3. Equation for e can be evaluated either at level 2 or 3 and its order against c or d equations is free. In practice, the propagation algorithm we implement has a *n log n* complexity.

3.2.2 Link of two Partial Orders

The approach allows an efficient link of two already sorted equation systems, to perform separated compilation, for instance. We don't need to launch the sorting algorithm from its initial step.
To link two equation systems, we only consider their common variables. In fact, as a consequence of equational semantics we rely on, we need to link an output variable of a system with an input one of the other and conversely. Assume that an output variable x of a sorted equation system A is an input variable of an another sorted equation system B. Let us denote $C^A(x)$ (resp $C^B(x)$) the CanDate of x in A (resp B). Similarly, we will denote $M^A(x)$ (resp $M^B(x)$) the MustDate of x in A (resp B). Then, we compute $\Delta_c(x) = |C^A(x) - C^B(x)|$, thus we shift by $\Delta_c(x)$ the CanDate of variables that depend of x in both equation systems (we look at the respective upstream partial orders computed for A and B equation systems). Similarly, we compute $\Delta_m(x) = |M^A(x) - M^B(x)|$, to shift the MustDate of variables that need x looking at the respective downstream partial orders of both equation systems.

3.3 Practical Issues

We have mainly detailed the theoretical aspect of our approach, and in this section we will discuss the practical issues we have implemented.

3.3.1 Effective compilation

We rely on the equational semantics to compile LE programs. To this aim we first compute the equation system associated with the body of a program (by computing the circuit associated with each node of the syntactic tree of the program). Doing that, according to the constructive approach we trust in, we also refine the value of signal and register status in the environment. Then, we translate each ξ circuit into a boolean circuit. To achieve this translation we define a mapping for encoding elements of ξ-algebra with a pair of boolean values. This encoding allows us to translate ξ equation system into a boolean equation system (each equation being encoded by two boolean equations). Finally, the effective compilation turns out to be the implementation of $\hookrightarrow$ propagation law to compute output and register values. We call the compilation tool that achieves such a task CLEM (Compilation of LE Module). In order to perform separate compilation of LE programs, we define an

internal compilation format called LEC (LE Compiled code). This format is highly inspired from the Berkeley Logic Interchange Format (BLIF [3]). This latter is a very compact format to represent netlists and we just add to it syntactic means to record the *CanDate* and *MustDate* of each variable defined by an equation. Practically, CLEM compiler, among other output codes, generates LEC format in order to reuse already compiled code in an efficient way (see section 3.2.2), thanks to the PERT method we implement.

3.3.2 Finalization Process

This approach to compile LE programs into a sorted ξ equation system in an efficient way requires to be completed by what we call a *finalization* phase. This latter is specific to our approach because the separated compilation implies to keep the $\perp$ status of signals for which no information has been propagated during the compilation phase, unlike the others synchronous approaches. A classical difficult problem in the compilation of synchronous languages is the "reaction to absence": signals are never set to absent and their final absent status is detected by a global propagation of signal status on the overall program. But these compilation algorithms prevent any separated compilation because they need a global vision of all status signals. In our approach, we keep the $\perp$ status of signal until the end of compilation and we drive back the absence resolution at the end of the process and we call it finalization. Hence, we replace all $\perp$ events by absent events. Notice that the finalization operation is harmless. The sorting algorithm relies on propagation of signal values, and the substitution of $\perp$ by 0 cannot change the resulting sorted environment.

3.3.3 Compilation scheme

Now, we detail the workflow we have implemented to specify, compile , simulate and execute LE programs. LE language help us to design programs. In the case of automaton, it can be generated by automaton editor like `galaxy` too. Each LE module is compiled in a LEC file and includes instances of RUN module references. These references can have been already compiled in the past by a previous call to the `clem` compiler. When the compilation phase is achieved, the finalization will simplify the final equation system and generates a file for a targeted usage: simulation, hardware description or software code. As mentioned in the introduction, an attractive consequence of our approach is the ability to perform validation of program behaviors. As a consequence, we can rely on model checking based tools to verify property of LE language. Thus, from lec files, we also generate `smv` files to feed the NuSMV model checker [4]. The `clem` workflow is summed up in the figure 2. All softwares belonging to the toolkit are available at: "http://www.unice.fr/L-EEA/gaffe/LE.html".

[3] `http://embedded.eecs.berkeley.edu/Research/vis`

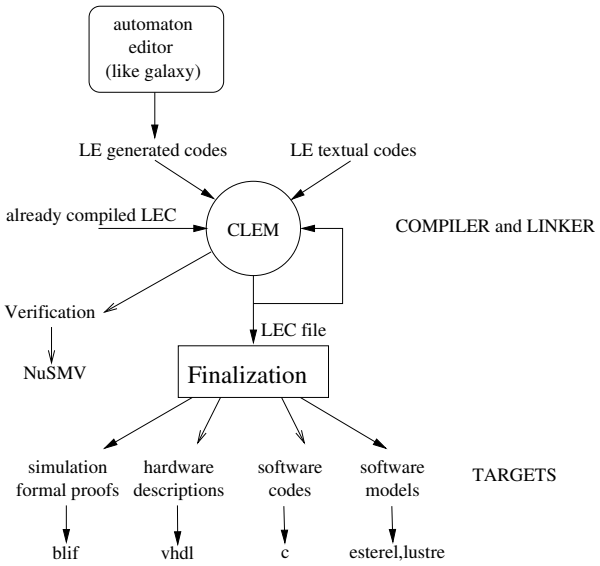


Fig. 2 Compilation Scheme

4 Related Works

The motivation for this work is the need for research on efficient and modular compilation of synchronous programs. Model-driven languages have been advocated to allow reliable and reusable system design. But rely on a responsive and modular semantics to compile a synchronous language is not at all straightforward since the causality problem must be faced. In [7], S Edwards and E Lee introduce a block diagram language to assemble different kinds of synchronous software. They introduce a fixpoint semantics inspired by the Esterel constructive semantics and remain deterministic even in the presence of instantaneous feedback. They propose exact and heuristic algorithms for finding schedules that minimize system execution time. But their block diagram language is an assembly language without hierarchy, while [13], consider hierarchical block diagrams. The purpose of this work is to generate code for "macros" (i.e a block diagram made of sub blocks) together with a set of interface functions in order to link with an external context. This idea is close to ours, but block diagrams are made of a hierarchy of sub diagrams interconnected and have not the expressiveness of LE language. Moreover, we can reload already compiled file dynamically in our compilation process. In [20], the authors consider a partial evaluation of Esterel programs: they generate distributed code that tries to compute as many outputs as possible while some inputs are unknown. In [16], a synchronous language Quartz is introduced and its compilation into a target job language. The separated compilation is done by splitting a program into sequential jobs corresponding to control flow locations of the program. This approach is well suited to generate distributed software. Similarly, some attempts allow a distributed

compilation of programs [19, 6] but they don't address the problem of dynamic link of already compiled sub programs.

5 Conclusion and Future Work

In this work, we introduced a synchronous language LE that supports a separated compilation. We defined its behavioral semantics giving a meaning to each program and allowing us to rely on formal methods to achieve verification. Then, we also defined an equational semantics to get a means to really compile programs in a separated way. Actually, we have implemented the `clem` compiler. This compiler is a link in the design chain we have to specify control-dominated process from different front-ends: a graphical editor devoted to automata drawing, or direct LE language specification to several families of back-ends.
In the future, we plan to improve our approach. The first improvement we aim at, is the extension of the language. To be able to deal with control-dominated systems with data (like sensor handling facilities), we will extend the syntax of the language. Then, we plan to integrate abstract interpretation techniques (like polyhedra intersection, among others) [5] to take into account data constraints in control. Moreover, we also need to discuss with signal processing or automation world through their specific tool Matlab/Simulink (http://www.mathworks.com). Another major improvement we are interesting in, concerns the development of verification means. Synchronous approach provides us with well-suited models to apply model checking techniques to LE programs. We already connect to the NuSMV model checker and provide with symbolic and bounded model-checking techniques application. A verification means successfully used for synchronous formalisms is that of observer monitoring [9]. We plan to introduce the ability to define safety properties as observers in LE and internally call NuSMV to prove them. Moreover, our modular approach opens new ways to perform modular model-checking. We need to prove that "assume-guarantee" technique [12] applies in our formalism.

References

1. C. André, H. Boufaïed, and S. Dissoubray. Syncharts: un modèle graphique synchrone pour système réactifs complexes. In *Real-Time Systems(RTS'98)*, pages 175–196, Paris, France, January 1998. Teknea.
2. G. Berry. *The Constructive Semantics of Pure Esterel*. Draft Book, available at: http://www.esterel-technologies.com 1996.
3. G. Berry. The Foundations of Esterel. In G. Plotkin, C. Stearling, and M. Tofte, editors, *Proof, Language, and Interaction, Essays in Honor of Robin Milner*. MIT Press, 2000.
4. A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV 2: an OpenSource Tool for Symbolic Model Checking. In Ed Brinksma and Kim Guldstrand Larsen, editors, *Proceeeding CAV*, number 2404 in LNCS, pages 359–364, Copenhagen, Danmark, July 2002. Springer-Verlag.

5. P. Cousot and R. Cousot. On Abstraction in Software Verification. In Ed Brinksma and Kim Guldstrand Larsen, editors, *Proceeeding CAV*, number 2404 in LNCS, pages 37,56, Copenhagen, Danmark, July 2002. Springer-Verlag.
6. S.A. Edwards. Compiling esterel into sequential code. In *Proceedings of the 7th International Workshop on Hardware/Software Codesign (CODES 99)*, pages 147–151, Rome, Italy, May 1999.
7. Stephen A. Edwards and Edward A. Lee. The semantics and execution of a synchronous block-diagram language. *Science of Computer Programming*, 48(1):21–42, July 2003.
8. N. Halbwachs. *Synchronous Programming of Reactive Systems*. Kluwer Academic, 1993.
9. N. Halbwachs, F. Lagnier, and P. Raymond. Synchronous observers and the verification of reactive systems. In M. Nivat, C. Rattray, T. Rus, and G. Scollo, editors, *Third Int. Conf. on Algebraic Methodology and Software Technology, AMAST'93*, Twente, June 1993. Workshops in Computing, Springer Verlag.
10. D. Harel and A. Pnueli. On the development of reactive systems. In *NATO, Advanced Study institute on Logics and Models for Verification and Specification of Concurrent Systems*. Springer Verlag, 1985.
11. C. Huizing and R. Gerth. Semantics of reactive systems in abstract time. In *Real Time: Theory in Practice, Proc of REX workshop*, pages 291–314. W.P. de Roever and G. Rozenberg Eds,LNCS, June 1991.
12. E. M. Clarke Jr., O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 2000.
13. R. Lublinerman and S. Tripakis. Modularity vs. reusability: Code generation from synchronous block diagrams. In *Design, Automation and Test in Europe (DATE08)*, 2008.
14. D. Potop-Butucaru and R. De Simone. *Formal Methods and Models for System Design*, chapter Optimizations for Faster Execution of Esterel Programs. Gupta, P. LeGuernic, S. Shukla, and J.-P. Talpin, Eds ,Kluwer, 2004.
15. Annie Ressouche, Daniel Gaffé, and Valérie Roy. Modular compilation of a synchronous language. Research Report 6424, INRIA, 01 2008. https://hal.inria.fr/inria-00213472.
16. Klaus Schneider, Jens Brand, and Eric Vecchié. Modular compilation of synchronous programs. In *From Model-Driven Design to Resource Management for Distributed Embedded Systems*, volume 225 of *IFIP International Federation for Information Processing*, pages 75–84. Springer Boston, January 2006.
17. Esterel Technologies. Esterel studio suite, www.estereltechnologies.com.
18. T.I.Kirkpatrick and N.R. Clark. Pert and an aid to logic design. *IBM Journal of Research and Develepment*, pages 135–141, March 1966.
19. D. Weil, V. Bertin, E. Closse, M. Poize, P. Venier, and J. Pulou. Efficient compilation of esterel for real-time embedded systems. In *Proceedings of the 2000 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems,*, pages 2–8, San Jose, California, United States, November 2000.
20. Jia Zeng1 and Stephen A. Edwards. Separate compilation for synchronous modules. In *Embedded Software and Systems*, volume 3820 of *Lecture Notes in Computer Science*, pages 129–140. Springer Berlin / Heidelberg, November 2005.