Towards a simple and safe Objective Caml compiling framework for the synchronous language SIGNAL

Zhibin Yang, Jean-Paul Bodeveix, M Filali

Official URL

DOI : https://doi.org/10.1007/s11704-017-6485-y

Towards a Simple and Safe Objective Caml Compiling Framework for the Synchronous Language SIGNAL

Zhibin YANG (✉)[1,2,3], Jean-Paul BODEVEIX (✉)[3], Mamoun FILALI (✉)[3]

1 School of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics,
Nanjing 211106, China
2 Collaborative Innovation Center of Novel Software Technology and Industrialization, Nanjing 211106, China
3 IRIT-CNRS, Université de Toulouse, Toulouse 31062, France

Abstract

This paper presents a simple and safe compiler, called MinSIGNAL, from a subset of the synchronous dataflow language SIGNAL to C, as well as its existing enhancements. The compiler follows a modular architecture, and can be seen as a sequence of source-to-source transformations applied to an intermediate representation which is named Synchronous Clocked Guarded Actions (S-CGA) and translation to sequential imperative code. Objective Caml (OCaml) is used for the implementation of MinSIGNAL. As a modern functional language, OCaml is adapted to symbolic computation and so, particularly suitable for compiler design and implementation of formal analysis tools. In particular, the safety of its type checking allows to skip some verification that would be mandatory with other languages. Additionally, this work is a basis for the formal verification of the compilation of SIGNAL with a theorem prover such as Coq.

Keywords Synchronous Languages, SIGNAL, Synchronous Clocked Guarded Actions (S-CGA), Objective Caml, Functional Programming

1 Introduction

Safety-critical systems are widely used in the fields of avionics, space systems, and nuclear power plants. Many of them are also considered as *reactive systems* [1], because they always continuously interact with their environment. The environment can be some physical devices to be controlled, a human operator, or other reactive systems. These systems receive from the environment input events, and compute the output information, which are finally returned to the environment. Synchronous programming is an important choice to design these systems, which relies on the *synchronous hypothesis* [3]. Firstly, the computation time is abstracted as zero, that lets system behaviors be divided into a discrete sequence of instants. At each instant, the system does input-computation-output, which takes zero time. Secondly, the different arrival time of events are abstracted as the relative order between events. Even if the physical time is abstracted, the inherent functional properties are not changed, so we can say this method focuses on functional behaviors at a platform-independent level.

There are several synchronous languages, such as ESTEREL [4], LUSTRE [5], SIGNAL [6], and QUARTZ [7], which can be considered as different implementations of the synchronous hypothesis. They adopt different programming styles, e.g., ESTEREL and QUARTZ have an imperative style suitable for control dominant applications while LUSTRE and SIGNAL respectively borrow functional and relational styles suitable for dataflow-oriented applications. Moreover, as a main difference from other synchronous languages, SIGNAL naturally considers a mathematical time model, in terms of a partial-order relation, to describe multi-clocked systems without the necessity of a global clock. This feature permits the description of globally asynchronous locally synchronous systems (GALS) [8, 9] conveniently.

E-mail: *yangzhibin*168@163.*com*, *bodeveix*@*irit*.*fr*, *filali*@*irit*.*fr*

The compilation process of synchronous languages is not limited to code generation: some analyses are first applied to determine if the specification is indeed executable. Let us mention the LUSTRE and ESTEREL causality analyses, the LUSTRE clock analyses and the ESTEREL constructive analysis. The SIGNAL compilation process contains two major analyses called *clock calculus* and *data-dependency graph construction* from which *code generation* directly follows. Moreover the clock calculus contains several steps, such as the synchronizations of each process, i.e., the generation of a set of equational constraints over clocks; the resolution of a system of clock equations; and the construction of a clock hierarchy on which the automatic code generation strongly relies.

In the SIGNAL compiler, the control flow expressed by abstract clocks serves to derive a control structure in automatic code generation. Thus, the quality of clock calculus has a strong impact on the correctness and efficiency of implementations. There are several optimizations of the clock calculus have been proposed to improve the quality of the code generated by the compiler, e.g., a code executed more efficiently or a code with smaller footprint. Optimizations aim at increasing the depth of the clock hierarchy in order to avoid useless tests in the generated code. They rely on discovering logical implications between boolean expressions, a conjunction being inserted as a descendent of one of its conjuncts. Several techniques have been considered: in the SIGNAL compiler-Polychrony [1] rewriting techniques tempt to normalize boolean or arithmetic expressions (with the option command *-crew*), [10] uses an SMT solver, [11] and [12] an interval-based data structure referred to as Interval-Decision Diagram (IDD).

In this paper, we propose an implementation of the main concepts of the SIGNAL compilation process which is encoded in the OCaml [2] language. In our compiler, called MinSIGNAL, we have considered existing enhancements such as [10], [11], and [12]. The compiler follows a modular architecture, and can be seen as a sequence of source-to-source transformations applied to an intermediate representation which is named Synchronous Clocked Guarded Actions (S-CGA) and translation to sequential imperative code.

(1) Why a new intermediate representation

Guarded commands [13], also called *asynchronous guarded actions* by J. Brandt et al. [14], are a well-established concept for the description of concurrent systems. In the spirit of the guarded commands, J. Brandt et al. propose *synchronous guarded actions* [15] as an intermediate representation for their QUARTZ compiler. As the name suggests, it follows the synchronous model. Hence, the behavior (control flow as well as data flow) is basically described by sets of guarded actions of the form $\langle \gamma \Rightarrow \mathcal{A} \rangle$. The boolean condition γ is called the guard and $\mathcal{A}$ is called the action. To support the integration of synchronous, polychronous and asynchronous models (such as CAOS [16] or SHIM [17]), they propose an extended intermediate representation, that is *clocked guarded actions* [14, 18] where one can declare explicitly a set of clocks. They also show how clocked guarded actions can be verified by symbolic model checking (SMV) and simulated by SystemC.

Compared with the existing SIGNAL compiler, we use clocked guarded actions as the intermediate representation, to integrate more synchronous languages such as QUARTZ, AIF[3] [14] into our compiler in the future. However, in contrast to the SIGNAL language, clocked guarded actions can evaluate a variable even if its clock does not hold. We mention also that the DC+ [19] intermediate format has been proposed as an intermediate format for compiling multiclock synchronous languages (ESTEREL, LUSTRE and SIGNAL). However, DC+ is introduced as a layer on top of DC which is a monoclock intermediate language. DC+ is characterized by a rich kernel with a monoclock guarded assignment (named `at`) and the equivalent of SIGNAL `when` and `default` constructs. Thus, we propose a variant of Clocked Guarded Actions, namely S-CGA, which constrains variable accesses as done by SIGNAL and where guarded assignments are multiclocked. Compared to DC+, the SIGNAL `when` and `default` are not part of S-CGA. Actually, they are compiled. The code generation from SIGNAL programs is adapted to the S-CGA context. In [20, 21], we have already defined the abstract syntax and the formal semantics of the S-CGA language.

(2) Why using OCaml in the tooling implementation

For a safety-critical system, the development process always follows strict guidelines. The development quality assurance applies as much to the final source code, as to the tools themselves. For instance, in the civil avionics the DO-178B/C airworthiness certification standard [22, 23] defines all the constraints ruling the aircraft software development. Moreover, one of the supplements to DO-178C, the DO-330 (Software Tool Qualification Considerations) provides a guidance to qualify tools. This means a tool, for example a compiler also needs to be qualified. For example, the SCADE

[1] http://www.irisa.fr/espresso/Polychrony

[2] http://ocaml.org/

[3] Averest Intermediate Format

SUITE [4] [5] KCG automatic code generator has been qualified as a development tool at DO-178B level A.

The choice of a programming language close and adapted to the tooling development is very important since a well-suited language leads to a simpler and safer way to encode the tooling requirements and consequently, a better and simpler traceability.

OCaml is a functional, imperative and object-oriented ML dialect. As a modern functional language, OCaml is adapted to symbolic computation and so, particularly suitable for compiler design and implementation of formal analysis tools. In particular, the safety of its type checking allows to skip some verification that would be mandatory with other languages, such as the memory allocation, coherency, initialization checks. The OCaml language has been used in Lucid Synchrone [24] (the Lustre language for reactive systems implementation), the compiler implementation of Prelude [25] (a synchronous language for critical embedded systems with multiple real-time constraints), and the Coq proof assistant [2] implementation. The first release of the SCADE KCG code generator was implemented in C and was available in 1999. Since 2005, Esterel Technologies has designed its new SCADE SUITE 6^{TM} in OCaml, which contains 65k lines of OCaml code [26]. As well, the abstract interpretation tool ASTREE [27] is also implemented in OCaml. The experiences of SCADE SUITE and of ASTREE by Airbus show that tools written in OCaml can be integrated in a critical software development process.

In this paper, the Ocaml implementation is a basis for the formal verification of the compilation of SIGNAL with a theorem prover such as Coq. Thus we restrict to the functional subset except for hash tables. Actually, the hash table can be replaced by the map structure in OCaml. Moreover, this work would be reused by ones interested in experimenting a new strategy for clock calculus and experimenting a new proof technique for the correctness of clock calculus (in a long term).

The rest of the paper is organized as follows. Section 2 gives an overview of the SIGNAL language. An analysis of the SIGNAL compilation process and the existing enhancements are given in Section 3. The MinSIGNAL is presented in Section 4. Section 5 discusses the related work, and Section 6 gives some concluding remarks and future work.

2 An Introduction to SIGNAL

In this section, we first introduce the basic concepts of the SIGNAL language such as signals and abstract clocks, then we give a sketch of its primitive constructs and a few extended constructs. Finally, a formal semantics of the primitive constructs is introduced.

2.1 Basic concepts

Signals. As declared in the synchronous hypothesis, the behaviors of a reactive system are divided into a discrete sequence of instants. At each instant, the system does input-computation-output, which takes zero time. So, the inputs and outputs are sequences of values, each value of the sequence being present at some instants. Such a sequence is called a *signal*. Consequently, at each instant, a signal may be present or absent (denoted by $\perp$). In SIGNAL, signals must be declared before being used, with an identifer (i.e., signal variable or the name of signal) and an associated type for their values such as integer, real, complex, boolean, event, string, etc.

Example 1 Three signals named *input1*, *input2*, *output* are shown as follows.

$$\begin{array}{lcccc} input1 & 1 & \perp & 3 & \perp \cdots \\ input2 & \perp & 5 & 7 & 9 \cdots \\ output & \perp & \perp & 10 & \perp \cdots \end{array}$$

Abstract Clock. The set of instants where a signal takes a value is the *abstract clock* of the signal. Two signals are synchronous if they are always present and absent at the same instants, which means they have the same abstract clock.

In the example given above, the abstract clock of *input1*, *input2* and *output*, denoted respectively $\widehat{input}1$, $\widehat{input}2$ and $\widehat{output}$, are defined by different sets of logical instants.

2.2 SIGNAL constructs

SIGNAL uses several constructs to express the relations between signals, including relations between values and relations between abstract clocks. Moreover, SIGNAL can specify the relations between the abstract clocks of signals in two ways: implicitly or explicitly.

Primitive Constructs. The primitive constructs can be classified into two families: monoclock operators (for which all signals involved have the same abstract clock) and multiclock operators (for which the signals involved may have different clocks).

[4] A successful commercial version of the synchronous language LUSTRE

[5] http://www.esterel-technologies.com/products/scade-suite/

- Monoclock operators, including *instantaneous function* and *delay*. The instantaneous function $x := f(x_1, \cdots, x_n)$ applied on a set of inputs $x_1, \cdots, x_n$ will produce the output x, while the delay operator $x := x_1 \$ \; init \; c$ sends the previous value of the input to the output with an initial value c.
- Multiclock operators, including *undersampling* and *deterministic merging*. The undersampling operator $x := x_1 \; when \; x_2$ is used to get an input at the true occurrence of another input, while the deterministic merging operator $x := x_1 \; default \; x_2$ is used to select between two inputs to be sent as the output, with a higher priority to the first input.

Notice that, these operators specify the relations between the abstract clocks of the signals in an implicit way.

In the SIGNAL language, the relations between values and the relations between abstract clocks of the signals, are defined as equations, and a *process* consists of a set of equations. Two basic operators apply to processes, the first one is the *composition* of different processes, and the other one is the *local declaration* in which the scope of a signal is restricted to a process.

Extended Constructs. SIGNAL also provides some operators to express control-related properties by specifying clock relations explicitly, such as clock synchronization, set operators on clocks (union, intersection, difference) and clock comparison.

- Clock synchronization, the equation $x_1 \,\hat{}{=}\, x_2 \,\hat{}{=} \cdots \hat{}{=} x_n$ specifies that signals $x_1, x_2, \cdots, x_n$ are synchronous.
- Set operators on clocks, such as the equation x:= x_1 ^ + x_2 defines the clock of x as the union of the clocks of signals x_1 and x_2, the equation x:= x_1 ^ * x_2 defines the clock of x as the intersection of the clocks of signals x_1 and x_2, the equation x:= x_1 ^ - x_2 defines the clock of x as the difference of the clocks of signals x_1 and x_2.
- Clock comparison, such as the statement x_1 ^ < x_2 specifies a set of inclusion relations between the clocks of signals x_1 and x_2, the statement x_1 ^ > x_2 specifies a set of containment relations between the clocks of signals x_1 and x_2, the statement x_1 ^ $\sharp$ x_2 specifies that the intersection of the clocks of signals x_1 and x_2 is empty.

2.3 A Trace Denotational Semantics

There exist several semantics for SIGNAL, such as denotational semantics based on traces (called trace semantics) [28–30], denotational semantics based on tags (called tagged model semantics) [29, 31], operational semantics presented in a structural style [6, 29], operational semantics defined by synchronous transition systems (STS) [**?**]. [33, 34] define a unified constructive semantic framework to unite QUARTZ and SIGNAL. This framework allows us to better understand the relationship between synchrony and polychrony. Here, we introduce the trace semantics. Moreover, the semantics of each of the extended constructs being defined in terms of the primitive constructs, we just consider the primitive constructs, that is core-SIGNAL. In [35], we have given a proof of the semantics equivalence between the trace semantics and the tagged model semantics of the core-SIGNAL.

In the following paragraphs, we first summarize the semantics domain i.e. the trace model, then the trace semantics.

(1) Trace Model

Let $\mathbf{X}$ be a set of signal variables, and let $\mathbf{V}$ be the set of values that can be taken by the variables. The symbol $\perp$ ($\perp \notin \mathbf{V}$) is introduced to express the absence of valuation of a variable. Then we denote:

$$\mathbf{V}^{\perp} = \mathbf{V} \cup \{\perp\}$$

Definition 1 (Signal) [29] A signal s is a sequence $(s_i)_{i \in I}$ of typed values (of $\mathbf{V}^{\perp}$), where I is the set of natural integers $\mathbf{N}$ or an initial segment of $\mathbf{N}$, including the empty segment.

A signal can be finite. However, we can extend the finite signal with infinite absences, to get an infinite one.

The definition of traces is given in the following paragraphs. Notice that, a signal is just a sequence of values corresponding to a signal variable, while a trace defines the synchronized sequences of values of a set of signal variables.

Definition 2 (Event) [28] Considering X a non-empty subset of $\mathbf{X}$, we call event on X any application

$$e : X \to \mathbf{V}_X^{\perp}$$

- $e(x) = \perp$ indicates that variable x has no value in the event.
- $e(x) = v$ indicates, for $v \in \mathbf{V}_x$, that variable x takes the value v in the event.

The *absent event* on X ($X \to \{\perp\}$), where all the signals are absent at a logical instant, is denoted $\perp_e(X)$. Moreover, the set of *events* on X ($X \to \mathbf{V}_X^{\perp}$) is denoted $\mathcal{E}_X$.

A *trace* is a sequence of events. For any subset X of $\mathbf{X}$, we consider the following definition of the set $\mathcal{T}_X$ of traces on $\mathbf{X}$.

Definition 3 (Traces) $\mathcal{T}_X$ is the set of traces on $\mathbf{X}$, defined as the set of applications $\mathbf{N} \to \mathcal{E}_X$ where $\mathbf{N}$ is the set of natural integers.

Similarly, a trace can be finite. However, we can extend the finite sequence with infinite absent events, to get an infinite trace.

Definition 4 (Sprocess) Given a SIGNAL process, its trace semantics, denoted as *Sprocess*, includes a set of signal variables defining the domain of the process and a set of traces.

(2) Trace semantics

Based on the trace model, the trace semantics of SIGNAL is presented as follows. It defines the set of traces associated to each primitive construct of SIGNAL.

Trace Semantics 1 The trace semantics of the instantaneous function $x := f(x_1, \cdots, x_n)$ is defined as follows:

$$\forall t \in \mathbb{N}$$
$$x_t = \begin{cases} \bot & if\ x_{1t} = \ldots = x_{nt} = \bot \\ f(x_{1t}, \ldots, x_{nt}) & if\ x_{1t} \neq \bot \wedge \ldots \wedge x_{nt} \neq \bot \end{cases}$$

At each instant t, the signals are either all present or all absent, i.e., they are synchronous, denoted $x \ \hat{} = x_1 \ \hat{} = \cdots \ \hat{} = x_n$. x_t gets the value of $f(x_{1t}, \ldots, x_{nt})$ when the signals are all present. The function f includes different mathematical operations, such as arithmetic operations, boolean operations, etc.

Trace Semantics 2 The trace semantics of the delay construct $x := x_1$ \$ *init c* is defined as follows:

$$– (\forall t \in \mathbb{N})\ x_{1t} = \bot \Leftrightarrow x_t = \bot$$
$$– \{k \mid x_{1k} \neq \bot\} \neq \emptyset \Rightarrow x_{\min\{k \mid x_{1k} \neq \bot\}} = c$$
$$– (\forall t \in \mathbb{N})\ x_{1t} \neq \bot \wedge \{k > t \mid x_{1k} \neq \bot\} \neq \emptyset$$
$$\Rightarrow x_{\min\{k > t \mid x_{1k} \neq \bot\}} = x_{1t}$$

Here, *min(X)* denotes the minimum of a non-empty set of naturals. Similarly to the instantaneous function, the delay construct also requires signals x and x_1 have the same clock, denoted $x \ \hat{}= x_1$. Given a logical instant t, x takes the most recent value of x_1 except the one at t. Initially, x takes the value c.

Trace Semantics 3 The trace semantics of the undersampling construct $x := x_1$ *when* x_2 is defined as follows:

$$\forall t \in \mathbb{N}$$
$$x_t = \begin{cases} x_{1t}\ if\ x_{2t} = true \\ \bot \quad otherwise \end{cases}$$

Here, x and x_1 have the same type and x_2 is a boolean signal. The clock of x is the intersection of the clock of x_1 and the true occurrences of x_2, denoted x=x_1 ˆ* $[x_2]$, where $[x_2] = \widehat{x_2} \wedge x_2$ represents the true occurrences of x_2.

Trace Semantics 4 The trace semantics of the deterministic merging construct $x := x_1\ default\ x_2$ is defined as follows:

$$\forall t \in \mathbb{N}$$
$$x_t = \begin{cases} x_{1t}\ if\ x_{1t} \neq \bot \\ x_{2t}\ otherwise \end{cases}$$

Here, signals x, x_1 and x_2 have the same type. The clock of x is the union of the clocks of x_1 and x_2, denoted $x = x_1$ ˆ+ x_2. Given a logical instant t, x_t gets the merge of the values of x_{1t} and x_{2t}, and the value of x_{1t} has a higher priority.

Finally the semantics of parallel composition is defined as the intersection of the semantics of the components. We apply these semantics rules to a SIGNAL process, to get a complete semantics of the process, that is *SProcess* (*Definition 4*).

Remark 1. The formal semantics is used in the programs transformations of the SIGNAL compiler, and it is also the basis of the validation of the compilation process.

3 An Analysis of the Main Concepts of the SIGNAL Compilation

In this section, we first present two formalizations, namely *clock algebra* and *conditional data-dependency graph*, which are used in the SIGNAL compilation process. Then, the main steps of the compilation are presented. Finally, we introduce the existing enhancements of the compilation process.

3.1 Formal models for SIGNAL program analysis

A SIGNAL program is a formal specification that is basically composed of equations describing relations of both values and clocks of the signals involved. The language allows one to mathematically reason on the properties of such a specification. One *clock algebra* associated to SIGNAL is the algebraic domain $\mathbb{Z}/3\mathbb{Z}$, the set of integers modulo 3 (i.e., $\mathbb{Z}/3\mathbb{Z} = \{-1, 0, 1\}$). Here 0 denotes a signal is absent, -1 means the signal is present and its value is false, and 1 means the signal is present and its value is true. As well, the reasoning approach for SIGNAL programs also includes *dependency graphs* to encode data dependencies.

(1) Clock algebra

In the SIGNAL context, relations over clocks are described using the clock algebra. The clock *cl* of a signal denotes a series of instants. The clock $\hat{x}$ of a signal x denotes the instants at which the signal x is present. The clock $[x]$ (resp. $[\neg x]$) denotes the instants at which x is present and holds the value *true* (resp. *false*). A clock expression e is either the empty clock, noted *0*, a signal clock *cl*, or the conjunction

$e1 \wedge e2$, the disjunction $e1 \vee e2$, the symmetric difference $e1 \backslash e2$ of two clock expressions e1 and e2.

$$cl ::= \hat{x} \mid [x] \mid [\neg x] \qquad (clock)$$

$$e ::= 0 \mid cl \mid e \wedge e \mid e \vee e \mid e \backslash e \qquad (clock\ expression)$$

Here, we have $[x] \vee [\neg x] = \hat{x}$ and $[x] \wedge [\neg x] = 0$.

(2) Conditional data-dependency graph

Intuitively, a data-dependency can be understood as the necessity that the value of some signal must be known in order to calculate the value of some other signal. A program is compiled into a graph that describes the data dependencies in the following sense: the edge

$$a \rightarrow^{cl} b$$

specifies that the computation of the node b, a signal or a clock, cannot be scheduled before that of the node a when the clock cl is present. A clock relation $cl = e$ specifies that the signal clock cl is present iff the clock expression e is true.

$$a, b ::= x \mid \hat{x} \qquad (node)$$

$$R ::= cl = e \mid a \rightarrow^{cl} b \qquad (data\ dependency)$$

Remark 2. At the semantics level, we mainly consider the relations between clocks and the relations between values. However, at the compiler level, we also need to consider the data-dependencies, because they determine the execution sequences in the generated code.

3.2 The main steps of the SIGNAL compilation

After a transformation from the user program (whose statements are expressed with both primitive constructs and extended constructs) to the normalized program whose statements are all expressed with primitive constructs, the SIGNAL compilation process contains two major analyses called *clock calculus* and *data-dependency graph construction* from which *code generation* directly follows. Moreover the clock calculus contains several steps, such as the synchronizations of each process, i.e., the generation of a set of equational constraints over clocks; the resolution of a system of clock equations; and the construction of a clock hierarchy on which the automatic code generation strongly relies.

(1) Synchronizations of SIGNAL processes

Each primitive construct inherently defines a relation over the clocks involved. Consequently, a system composed out of multiple processes describes a system of these relations, which is, an equation system of clock variables that describe the synchronizations of the system. Based on the clock algebra presented in section 3.1, for each primitive construct, the clock synchronizations are given by Table 1. Moreover, the clock synchronizations can be described equivalently using SIGNAL extended constructs (e.g., $x\hat{}= x_1\hat{}+ x_2$).

Table 1 Synchronizations of each primitive construct

P	*synchronizations of* P *in clock algebra*	*synchronizations* *of* P *in SIGNAL*
$x := f(x_1, \cdots, x_n)$	$\hat{x} = \hat{x}_1 = \cdots = \hat{x}_n$	$x\hat{}= x_1 \mid ... \mid x\hat{}= x_n$
$x := x_1\ \$\ init\ c$	$\hat{x} = \hat{x}_1$	$x\hat{}= x_1$
$x := x_1\ when\ x_2$	$\hat{x} = \hat{x}_1 \wedge [x_2]$	$x\hat{}= x_1\hat{}*\ when\ x_2$
$x := x_1\ default\ x_2$	$\hat{x} = \hat{x}_1 \vee \hat{x}_2$	$x\hat{}= x_1\hat{}+ x_2$

(2) Solving the clock equation system

The SIGNAL philosophy strongly emphasizes that clocks indicate the control of data-flow specifications. Accordingly the control-flow of the target executable code is synthesized from relations over clocks, or synchronizations, that is, at any given instant, before the value of a signal x is computed, a test must be made on the presence/absence of x, namely the presence/absence of its clock $\hat{x}$. So there is a need for a resolution method that will allow to efficiently check the presence of a clock. For that purpose, the clock equation system to be solved is extracted from the system by applying the rules of Table 1. The general approach for solving this equation system is given by *triangularization* [36]:

The equational system is transformed into an ordered set of so called *directed definitions*, that is, a system of equations of the form $h = h_1 \langle op \rangle h_2$, where h is a newly defined clock, h_1, h_2 are previously defined clocks and $op \in \{\wedge, \vee, \backslash\}$ is some operator on clocks. This representation ensures the absence of clock-to-clock cycles. Several problems should be solved to get this representation:

- *Multiple definitions*: If the equation system contains more than one equation with clock h on the left side, the equality of their respective right sides must be proven.
- *Clock-to-clock cycles*: If the equation system contains cyclic dependencies, they have to be eliminated.
- *Complex relations*: If equations are not of the form $h = h_1 \langle op \rangle h_2$, e.g., given by $h_1 \langle op \rangle h_2 = k_1 \langle op \rangle k_2$, an attempt can be made to prove the equivalence of the formulas with rewriting techniques.

The SIGNAL compilation process involves solving these problems, mainly by using a rewriting system plus some heuristics, in order to achieve triangular form. But since these problems are complex, the compiler is not complete: If no appropriate rewriting rule can be applied, an input program may be rejected although the system could be solved. In any case, if some equalities cannot be proven or if some cycle cannot be eliminated, an input program is considered temporally incorrect and rejected.

(3) Hierarchical representation of clock equations

Efficient code generation requires a "good" representation of the solutions of the clock equations which result from the previous step. In order to efficiently apply rewriting and keep track of the triangularity (i.e., triangularization preservation) of the equation systems, a tree-based representation of the equations is used, named the *clock hierarchy*. Moreover, the paper [9] discusses the link between perfect synchrony and asynchrony, and shows that the *endochrony* property of a clock hierarchy is a sufficient condition to get executable and deterministic code from a given clock hierarchy.

The synthesis of clock hierarchy in SIGNAL programs relies on an efficient algorithm [36] that has been implemented in the compiler. We summarize it as follows:

- For a Boolean signal x, the known partition $[x] \subseteq \hat{x} \supseteq [\neg x]$ is represented by a basic partition tree, whose edges are the set inclusion relations.
- The representation of the whole equation system of synchronizations starts with representing all directed definitions by a forest of clock trees.
- Then, two clock trees are iteratively fusioned by inserting one tree into another: Let *T*, *T'* be two clock trees with roots r and h respectively, where h is defined by the directed equation $h = h_1 \langle op \rangle h_2$, and h_1 and h_2 are subtrees of *T*. Then the fusion of *T'* into *T* is applied by adding the tree *T'* to the immediate children of the least common ancestor of h_1 and h_2 in *T*. This way, the subtree *T'*, which is defined by the operands h_1 and h_2, is placed directly under the least common ancestor of these operands. This preserves the structural property of triangularization and gives credit to clock inclusion relations which yield more efficient nested if-tests.

Intuitively, the clock of a node is a subset of the clock of its parent. The algorithm repeatedly tries to rewrite the equations so that they match the criteria of the fusion step and then executes the fusion until this cannot be done anymore.

As shown in the report [36], we can do optimizations on the clock hierarchy to improve the quality of the code generated by the compiler, e.g., a code executed more efficiently or a code with smaller footprint. For instance, the resulting tree-based representation of the equation systems over clocks can be optimized in the sense that the insertion step during the fusion chooses a parent with greatest depth.

(4) Code generation

The code generation is based on the transformations presented in the previous sections. It is strongly guided by the clock hierarchy resulting from the clock calculus to structure the target language program, and by the conditional data-dependency graph not only to locally order elementary operations in sequences, but also to schedule component activations in a hierarchical target code. This code can be obtained in different target languages, among which the most used are C, C++, and Java.

When a SIGNAL program P is proved to be *endochronous*, the generation of its associated code is straightforward. Each node of the clock tree corresponding to P is characterized by a Boolean expression that expresses a condition (*if-then-else* structures). The statements that depend on each node are computed whenever the associated condition is evaluated to be true, meaning that the expressed clock is present. On the other hand, the code generation takes into account the conditional dependency graph that characterizes the order according to which statements are to be computed at the same clock instants.

Furthermore, the code generated especially the nesting of *if-then-else* structures can be optimized, if h and k are two clocks such that $h \subseteq k$, then for an instant t, the following clock implication holds:

$$t \notin k \Longrightarrow t \notin h$$

In others words, if the test $t \in k$ fails, there is no need to test if $t \in h$. Thus, code generation can take advantage of the clock inclusion relation between clocks.

As an example, the code

```
$if\ present(k)\ then$
    $do - action - k$
    $if\ present(h)\ then$
        $do - action - h$
    $endif$
$endif$
```

is more efficient than the code

```
if present(k) then
   do − action − k
endif
if present(h) then
   do − action − h
endif
```

3.3 Handling Numerical Expressions

For the under-sampling construct, remember that the clock of the Boolean expression x is partitioned into $[x]$ and $[\neg x]$, which are referred to as *condition-clocks*. In the SIGNAL compiler, if x is defined by a numerical expression such as an integer comparison, $[x]$ and $[\neg x]$ are seen as black boxes by default. As a consequence the relation in the numerical expression is unknown and useless comparisons are generated.

Here, we use an example to explain the handling of numerical expressions.

Example 2 A process named *Numerical_Expr* is given:

```
process Numerical_Expr =
(? integer a, b, x1;
   boolean c1, c2;
 ! integer x, y, z1, z2, z3, m, s1, s2;
)
(| a^= b^= x1^= c1^= c2
 | x := a $ init 1
 | y := b $ init 2
 | z1 := x when (a > 0)
 | z2 := x when (a <= 0)
 | z3 := y when (b > 0)
 | m := z1 default z2 default z3
 | s1 := x1 when c1
 | s2 := x1 when (c1 and c2)
 |);
```

- Default Polychrony compiler

A part of the result of the clock calculus generated by the compiler in Polychrony (without the option *-crew*) is shown as follows. Here, $[a > 0]$, $[not\ (a > 0)]$, $[a <= 0]$, $[not\ (a <= 0)]$, $[b > 0]$, and $[not\ (b > 0)]$ are considered as black boxes. So, the compiler treats $[a > 0]$ and $[not\ (a <= 0)]$, $[not\ (a > 0)]$ and $[a <= 0]$, as different clock equivalence classes. Actually they are in the same clock equivalence class respectively. It follows that the compiler cannot analyze some static properties of a program, such as clock exclusion or clock emptiness, since numerical expressions are not suitably abstracted.

```
| (| CLK := when (a>0)
   | CLK_13 := when (not (a>0))
   |)
| (| CLK_14 := CLK ^* CLK_z1 |)
| (| CLK_z1 := CLK_a ^* CLK
   | CLK_z1 ^= z1
   | (| z1 := x when CLK_z1 |)
   |)
| (| CLK_17 := when (a<=0)
   | CLK_18 := when (not (a<=0))
   |)
| (| CLK_19 := CLK_17 ^* CLK_z2 |)
| (| CLK_z2 := CLK_a ^* CLK_17
   | CLK_z2 ^= z2
   | (| z2 := x when CLK_z2 |)
   |)
| (| CLK_22 := when (b>0)
   | CLK_23 := when (not (b>0))
   |)
| (| CLK_24 := CLK_22 ^* CLK_z3 |)
| (| CLK_z3 := CLK_a ^* CLK_22
   | CLK_z3 ^= z3
   | (| z3 := y when CLK_z3 |)
   |)
```

A part of the C code generated by Polychrony compiler is shown as follows. It also appears that the test C_s2 associated to $[c1\ and\ c2]$ is performed even if $[c1]$ or $[c2]$ tests fail. It means that the clock inclusion property $[c1\ and\ c2] \Rightarrow [c1]$ has not been detected. If $[c1\ and\ c2]$ had been placed as a son of one of $[c1]$ or $[c2]$ while preserving "arborescent canonical form", the test of the conjunction would have been performed only when one of the argument is true. However, as mentioned in section 3.2, such optimizations could be performed by the C compiler itself.

```
C_z1 = a > 0;
C_z2 = a <= 0;
C_z3 = b > 0;
C_s2 = c1 && c2;
C_CLK_52 = C_z1 || C_z2;
C_m = C_CLK_52 || C_z3;
if (c1)
     { s1 = x1; w_exp1_s1(s1);
     }
if (C_z1)
     { z1 = x; w_exp1_z1(z1);
     }
if (C_z2)
     { z2 = x; w_exp1_z2(z2);
     }
if (C_z3)
     { z3 = y; w_exp1_z3(z3);
     }
if (C_s2)
     { s2 = x1; w_exp1_s2(s2);
     }
if (C_m)
     {if (C_z1)
        m = z1;
     else if (C_z2)
        m = z2;
        else
            m = z3;
        w_exp1_m(m);
        }
```

- Compilation with the option *-crew*

Even if the SIGNAL compiler does not fully handle numerical expressions, it is possible to let it do some rewriting of numerical conditions as Boolean expressions by using the option *-crew*. A part of code generated for the Example 2 is given as follows. With this option, comparisons are rewritten using only the operators == (for example $CLK\_81 = 0 == a$) and <= (for example $CLK\_83 = 0 <= a$). Thus, $[a > 0]$ and $[a <= 0]$ can be rewritten as $C\_z1 = !CLK\_81 \&\& CLK\_83$ and $C\_z2 = CLK\_81 || !CLK\_83$ respectively. As we can see, the rewriting of comparisons permits a more precise analysis on clocks.

```
    CLK_81 = 0 == a;
    CLK_83 = 0 <= a;
    C_z1 = !CLK_81 && CLK_83;
    C_z2 = CLK_81 || !CLK_83;
    C_z3 = (0 <= b) && !(0 == b);
    C_s2 = c1 && c2;
```

- Enhancement with a combined numerical-Boolean abstraction

Paul Feautrier et al [10] propose a combined numerical-Boolean abstraction to deal with these problems. In the new abstraction, every signal in a program is associated with a pair of the form (clock, value), where clock is a Boolean function and value is a Boolean or numeric function. They also use a SMT solver to reason on the new abstraction, and get more precise clock analysis because they treat the numerical expression as white boxes.

4 Inside the MinSIGNAL Compiler

In this section, we present the OCaml code of the compilation process of MinSIGNAL. First, the modular architecture of MinSIGNAL is given in a global view. Second, the compilation progress is presented step by step.

4.1 A Modular Architecture

As shown in Figure 1, the MinSIGNAL compiler has a modular architecture, and can be seen as a sequence of source-to-source transformations applied to the intermediate representation S-CGA and then translation to sequential imperative code. Specifically, the compilation process is mainly structured as five modules. At each module, there are several sub-modules.

- Module 1: Normalization of the user program. Beyond the usual lexical analysis, parsing and type checking, the compiler will transform the user program (using the subset of SIGNAL) whose statements are expressed with both primitive constructs and extended constructs to the normalized program whose statements are all expressed with primitive constructs.
- Module 2: Synchronizations of SIGNAL processes. As a difference with the existing SIGNAL compiler, we construct S-CGA from the normalized program. S-CGA contains control flow (the relations between clocks) as well as data flow (the relations between values).
- Module 3: Solving the clock equation system. If the system of clock equations contains more than one equation with the same clock, the execution of the generated code will check the same control condition several times, and it is inefficient. This is why we need to resolve it. We can use BDD or SMT technology to check the equivalence of two clock equations, and put the corresponding clock variables into the same equivalence class. We also check the *endochrony* property at this step, namely there is just one master clock.
- Module 4: Hierarchical representation of clock equations. The code generation is based on both the clock hierarchy and the data dependencies. However, there may be clock-to-data cycles. To reduce these cycles, we first sort all the guarded actions. It will be easier to construct a clock hierarchy based on deterministic sorting, and we consider the sorting as a depth first search (DFS) order.
- Module 5: Code generation and optimization. The basic idea of code generation is the same as in the SIGNAL compiler. Furthermore, we do some optimizations at the code level. Given two equations such as $y = x\ when\ x_1$ and $z = x\ when\ (x_1\ and\ x_2)$, there is a clock-inclusion relation: $[x_1 \wedge x_2] \rightarrow [x_1]$, i.e., the clock of $[x_1 \wedge x_2]$ is a subset of the clock of $[x_1]$. Consequently, we can do the code optimization illustrated as follows. If control condition x_1 holds, we do not need to check x_1 again in $x_1 \&\& x_2$. We just need to check if x_2 holds or not.

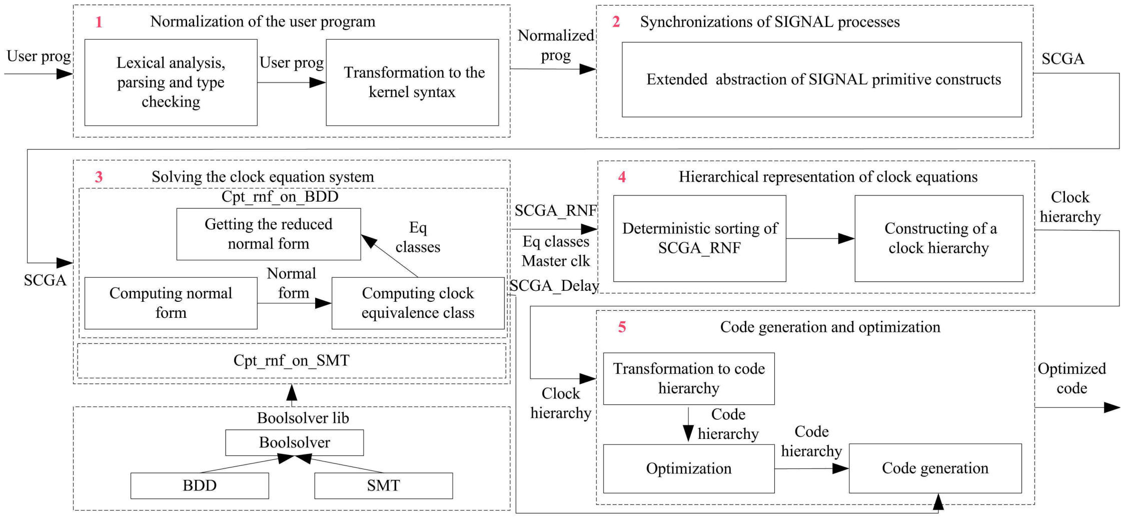
1 Normalization of the user program
User prog
Lexical analysis, parsing and type checking
User prog
Transformation to the kernel syntax
Normalized prog
2 Synchronizations of SIGNAL processes
Extended abstraction of SIGNAL primitive constructs
SCGA
3 Solving the clock equation system
Cpt_rnf_on_BDD
Getting the reduced normal form
Eq classes
Normal form
Computing normal form
Computing clock equivalence class
SCGA
Cpt_rnf_on_SMT
Boolsolver lib
Boolsolver
BDD
SMT
SCGA_RNF
Eq classes
Master clk
SCGA_Delay
4 Hierarchical representation of clock equations
Deterministic sorting of SCGA_RNF
Constructing of a clock hierarchy
Clock hierarchy
5 Code generation and optimization
Transformation to code hierarchy
Clock hierarchy
Code hierarchy
Code hierarchy
Optimization
Code generation
Optimized code

Figure 1 A modular architecture for the formalization of the SIGNAL compilation process

```
if (x_1){                         if (x_1){
   do actions                        do actions
   ...                               ...
   if (x_1&&x_2){       ⇒            if (x_2){
    do actions                        do actions
    ...}                              ...}
}                                 }
```

As mentioned before, the safety of the type checking of OCaml allows to skip some verification that would be mandatory with other languages. Moreover, the OCaml code is compact which allows less verification efforts. MinSIGNAL consists in about 3000 lines of OCaml code (Table 2).

Table 2 The OCaml code size of MinSIGNAL

Module name	OCaml code
Normalization of the user program	750 lines
Intermediate representation S-CGA	350 lines
Solving the clock equation system	750 lines
Hierarchical representation of clock equations	550 lines
Code generation and optimization	500 lines

4.2 Normalization of the user program

As shown in section 2.2, the definition of the extended constructs is derived from a combination of the primitive constructs to provide the user with suitable *macros*. So, beyond the usual lexical analysis, parsing and type checking, the compiler will transform the user program whose statements are expressed with both primitive constructs and extended constructs to the normalized program whose statements are all expressed with primitive constructs.

(1) Lexical analysis, parsing and type checking

The lexical analysis and parsing of MinSIGNAL are implemented with standard tools, OCamlLex and OCamlYacc. As usual, they translate a string of characters to a sequence of tokens and then to an abstract syntax tree, which is necessary for any complex program manipulation. We don't give the details of the lexical analysis, parsing and type checking, because of the space limitations.

Here, we give a simplified abstract syntax which is defined in OCaml. *ty* represents the data types of SIGNAL such as *integer*, *real*, *complex*, *boolean*, etc. A SIGNAL specification *spec* is a set of processes. A process *proc* is a set of equations (*eqn list*) for signals specifying relations between values, on the one hand, and clocks, on the other hand, of the signals involved. Moreover, a process also includes its name, input signals, output signals and local declarations. An equation can denote the relations between values or the relations between clocks, *Ident* represents the identifier of a signal, *CInt*, *CReal*, *CBool*, *CString*, and *CChar* represents constants defined in the equations, *Func*, *Delay*, *When*, and *Default* denote the primitive constructs, set operators on clocks such as ^+, ^* and ^- are also represented in *Func*, while clock comparison operators such as ^<, ^>, ^♯ and ^= are represented in *Constraint*.

```
type ty = TBasic of string
type exp =
  | Ident of string
  | CInt of int
  | CReal of float
  | CBool of bool
  | CString of string
  | CChar of char
  | Func of string * exp list
  | Delay of exp * exp
  | When of exp * exp
  | Default of exp * exp
type eqn =
  | Ass of string * exp
  | Constraint of string * exp * exp
type decl = ty * string
type proc = Proc of string*(decl list)
            *(decl list)*(eqn list)*(decl list)
type spec = proc list
```

(2) Transformation of the user syntax to the kernel syntax

We first give the kernel syntax (corresponding to the core-SIGNAL) which is defined in OCaml. The abstract syntax of core-SIGNAL, of clock equation (in section 4.4), and of reduced normal form (i.e., equivalence class in section 4.5) have a common style, thus we use a type *kind* with three values *Ksignal*, *Clock*, and *Class*. In the user syntax, the expression can be iterated, while in the kernel syntax, the expression has been flattened.

```
type kty= KTBasic of string
type const =
    | SCInt of int
    | SCReal of float
    | SCBool of bool
    | SCString of string
    | SCChar of char
type kind=
    | Ksignal
    | Clock
    | Class of Quotient.t
type var = string * kind
type sexp =
    | SVar of var
    | SConst of const
    | SFunc of string * var list
type kexpSig =
    | KDelay of string * sexp
    | KWhen of string * string
    | KDefault of string * string
type keqn =
    | KAss of var * sexp
    | KAssSig of string * kexpSig
type kdecl = kty * string
type kproc = KProc of string*(kdecl list)
             *(kdecl list)*(keqn list)
type kspec = kproc list
```

Several rules are applied to the transformation from the extended constructs to the primitive constructs.

Transformation 1. We have one basic rule: the equation $clk := \hat{x}$ can be expressed using primitive constructs as follows.

Clock extraction	Corresponding primitive constructs
$clk := \hat{x}$	$clk := (x = x)$

Transformation 2. The set operations on abstract clocks such as clock intersection, union, and difference, can be expressed by the following constructs, and then can be expressed using primitive constructs.

Set operations on clocks	Corresponding constructs
$x := x_1 \hat{}* x_2$	$x := \hat{x_1} \quad when \quad \hat{x_2}$
$x := x_1 \hat{}+ x_2$	$x := \hat{x_1} \quad default \quad \hat{x_2}$
$x := x_1 \hat{}- x_2$	$x :=$ $when\ ((not\ \hat{x_2})$ $default\ \hat{x_1})$

Notice that $x := when\ ((not \quad \hat{x_2}) \quad default \quad \hat{x_1})$ can be rewritten as $x := ((not \quad \hat{x_2}) \quad default \quad \hat{x_1})\ when\ ((not \quad \hat{x_2})$ $default \quad \hat{x_1})$

Transformation 3. The comparison operations of abstract clocks can be expressed using set operations on clocks, and then can be expressed using primitive constructs. Here, $\hat{0}$ specifies the empty clock.

Clock comparison operations	Set operations on clocks
$x_1 \hat{}< x_2$	$x_1 \hat{}= x_1 \hat{}* x_2$
$x_1 \hat{}> x_2$	$x_1 \hat{}= x_1 \hat{}+ x_2$
$x_1 \hat{}\sharp x_2$	$\hat{0} \hat{}= x_1 \hat{}* x_2$
$x_1 \hat{}= x_2$	$clk := \hat{x_1} = \hat{x_2}\ where\ clk$

Dealing with constant signals. In addition, we'd like to see the clock equation system defined in clock algebra directly. There is a little problem about the presence of constant signals. For example, given a process with two equations of signals,

$$y = 2\ when\ b$$
$$y' = 2\ when\ b'$$

we have the clock relations as follows,

$$\hat{y} = \hat{2} \wedge [b]$$
$$\hat{y'} = \hat{2} \wedge [b']$$

However, the clock $\hat{2}$ in $\hat{y} = \hat{2} \wedge [b]$ and in $\hat{y'} = \hat{2} \wedge [b']$ may be not the same. Namely, the presence of constant signals depend on their usage context. To avoid conflicts, we use a new signal variable (defined in the local declaration) to replace the constant signal, and the new signal variable will be defined using a *delay* construct, for example $x_1 = x_1\ \$\ init\ 2$

and $x_2 = x_2$ \$ *init* 2. Then, the process will be transformed as,

$$\begin{aligned} y &= x_1 \ when\ b, \\ y' &= x_2 \ when\ b' \\ x_1 &= x_1 \ \$\ init\ 2 \\ x_2 &= x_2 \ \$\ init\ 2 \end{aligned}$$

Finally, we get the clock relations as follows,

$$\begin{aligned} \hat{y} &= \hat{x_1} \wedge [b] \\ \hat{y'} &= \hat{x_2} \wedge [b'] \end{aligned}$$

Notice that, we compute the data-type of each new signal variable by type inference.

Based on the kernel syntax and previous transformation rules, the user program will be transformed into the normalized program and then will be flattened.

The comparison operations of abstract clocks are transformed into the set operations on clocks. Its OCaml code is given as follows.

```
let rec elim_ctrs_in_eqns eqns =
 List.concat (List.map (
 function
   | Ass (s, e) as eq -> [eq]
   | Constraint (s, e1, e2) ->
     begin match s with
     |"^<"->
        [Constraint("^=",e1,Func("^*",[e1;e2]))]
     |"^>"->
        [Constraint("^=",e1,Func("^+",[e1;e2]))]
     |"^#"->
        [Constraint("^=",Func("^",[CInt(0)]),
         Func("^*",[e1;e2]))]
     |"^="->
        let c1=new_var "C" (TBasic "event")
        and c2=new_var "C" (TBasic "event")
        and c3=new_var "C" (TBasic "event")
        in
        if isIdentorConst(e2) then
        [Ass(c1, Func("==",[e1;e1]));
        Ass(c2, Func("==",[e2;e2]));
        Ass(c3, Func("==",[Ident c1;Ident c2]))]
        else
        let c4=new_var "C" (TBasic "event") in
        [Ass(c1, Func("=",[e1;e1]));
        Ass(c2,e2);
        Ass(c3, Func("=",[Ident c2; Ident c2]));
        Ass(c4, Func("=",[Ident c1; Ident c3]))]
    |_->failwith("NYI:unknown␣constraint")
   end
 ) eqns )
```

For the set operations on abstract clocks, they are firstly transformed into corresponding constructs, which are implemented by the constructors *Func("ˆ+",[e1;e2])*, *Func("ˆ*",[e1;e2])* and *Func("ˆ-",[e1;e2])* in the recursive function *flatten*. Then, all the clocks such as $\hat{x}$ will be expressed using primitive constructs, that is the constructor *Func("ˆ",[e])*. Finally, all the primitive constructs transformed will be flattened.

```
let rec flatten = function
 |Func ("^+",[e1;e2]) ->
 flatten(Default(Func("^",[e1]),Func("^",[e2])))
 |Func ("^*",[e1;e2]) ->
  flatten(When(Func("^",[e1]),Func("^",[e2])))
 |Func ("^-",[e1;e2]) ->
 let e3 = Default(Func("not", [Func("^",[e2])]),
  Func("^",[e1])) in
  flatten (When(e3,e3))
 |Func("^", [e]) ->
  let v1=new_var "CL"(TBasic "event") in
  let e1=flatten e in
  add_eqn(Ass(v1, Func(":=",[e1;e1])));
  Ident v1
 |Func(f,l) as e ->
  let v1=new_var "F" (TBasic(get_type e)) in
  add_eqn(Ass(v1, Func(f,List.map flatten l)));
  Ident v1
 |Delay(e1,e2) as e ->
  let v2=new_var "D" (TBasic(get_type e)) in
  add_eqn(Ass(v2, Delay(flatten e1, e2)));
  Ident v2
 |Default(e1,e2) as e ->
  let v3=new_var "M" (TBasic(get_type e)) in
  add_eqn(Ass(v3,Default(flatten e1, flatten e2)));
  Ident v3
 |When(e1,e2) as e ->
  let v4=new_var "W" (TBasic(get_type e)) in
  add_eqn(Ass(v4, When(flatten e1, flatten e2)));
  Ident v4
 |Ident _ as e -> e
 |e (*this is constant*)->
  let k=new_var "K" (TBasic(get_type e)) in
  add_eqn(Ass(k, Delay(Ident k, e)));
  Ident k
```

4.3 S-CGA Intermediate representation

In papers such as [18], clocked guarded actions have been defined as a common representation for synchronous (via synchronous guarded actions), polychronous and asynchronous (via asynchronous guarded actions) models. It has a multi-clocked feature. However, in contrast to the SIGNAL language, clocked guarded actions can evaluate a variable even if its clock does not hold [18] [20]. In this case the read value is the most recently written value, while in SIGNAL *read and writes can be simultaneous provided the causality is respected.* As a consequence, we have introduced an intermediate representation S-CGA [20, 21], which is a variant of clocked guarded actions. In one hand, S-CGA expresses the relations between signals, including the relations between abstract clocks and the relations between values, which are consistent with the SIGNAL program. In the other hand, as an intermediate representation, S-CGA permits us to integrate more synchronous languages such as QUARTZ, AIF into our compiler in the future.

(1) The definition of S-CGA

S-CGA has the same structure as clocked guarded actions, but has *different semantics*.

Definition 6 (S-CGA) An S-CGA system is represented by a set of guarded actions of the form $\langle\gamma \Rightarrow \mathcal{A}\rangle$ defined over a set of variables X. The Boolean condition γ is called the guard and $\mathcal{A}$ is called the action. Guarded actions can be of the following forms:

$$\begin{array}{lll}(1) & \gamma \Rightarrow x = \tau & \text{(immediate)}\\ (2) & \gamma \Rightarrow next(x) = \tau & \text{(delayed)}\\ (3) & \gamma \Rightarrow \ assume(\sigma) & \text{(assumption)}\end{array}$$

where

- the guard γ is a Boolean condition over the variables of X, and their respective clocks. For a variable $x \in X$, we denote:
  - its clock $\hat{x}$,
  - its initial clock $init(\hat{x})$ as the clock which ticks the first time (if any) where $\hat{x}$ ticks.
- τ is an expression over X,
- σ is a Boolean expression over the variables of X and their clocks.

An immediate assignment $x = \tau$ writes the value of τ immediately to the variable x. The form (1) implicitly imposes that if γ is defined[6] and its value is true, then x is present and τ is defined.

A delayed assignment $next(x) = \tau$ evaluates τ in the given instant but changes the value of the variable x at next time clock $\hat{x}$ ticks.

The form (3) defines a constraint. It determines a Boolean condition which has to hold when γ is defined and true. All the execution traces must satisfy this constraint. Otherwise, they are ignored.

Guarded actions are composed by using the parallel operator $||$.

A SIGNAL process includes its name, input signals, output signals, local declarations, and equations which denote the relations between clocks or the relations between values. In the OCaml code, we consider also inputs and outputs as guarded actions in S-CGA. The *immediate* guarded action is split into $init(\hat{x}) \Rightarrow x = \tau$ and $\gamma \Rightarrow x = \tau$ where $init(\hat{x})$ is not in the guard γ. The *assumption* guard action is also defined in the type *action*. We use another hash table *ddes* to deal with the *delay* guarded action.

The corresponding OCaml code is given as follows.

```
(*immediate, assumption,*)
(*input, and output guarded actions*)
type action=
  Dass of sexp * var * sexp
   |Input of var * string
   |Output of string
let action2str = function
   |Dass(se1, sv2, se2) ->
     sexp2str se1 ^ "=>" ^ var2str sv2
     ^ "=" ^ (sexp2str se2)
   |Input (sv, s) ->
     var2str sv ^ "=>" ^ "Read" ^ s
   |Output s ->
     "Write" ^ s
(*init part of immediate guarded actions*)
let dces=Hashtbl.create 1000
let hash2str1 tls= Hashtbl.fold (fun k v r ->
   r ^ "\n" ^ "init(" ^ sexp2str k ^ ")"
   ^ "=>" ^ keqn2str v ) tls ""
(*delay guarded actions*)
let ddes=Hashtbl.create 1000
let rec v2str = function
  |KAss ((sv,k),e) ->
    "next("^sv ^")" ^"="^(sexp2str e)
  |_ -> failwith"v2str"
let hash2str2 tls=Hashtbl.fold(fun k v r ->
  r^"\n"^sexp2str k^"=>"^v2str v) tls ""
```

(2) From normalized program to S-CGA

Clock synchronization constraints are given in Table 3. Except for the under-sampling construct, we also enhance the clock synchronization of the deterministic-merging construct. The construct $x := x_1\ default\ x_2$ will be split as two assignments: $x = x_1$ corresponding to the clock $\widehat{x_1}$ and $x = x_2$ corresponding to the clock $\widehat{def\_\_x_2} = \widehat{x_2} \backslash \widehat{x_1}$. Namely, a new clock variable $\widehat{def\_\_x_2}$ is added. This enhancement will be useful in the construction of the clock hierarchy to avoid clock-to-data cycles (section 4.5).

Table 3 Extended synchronizations of each primitive construct

P	*synchronizations of* P *in extended clock abstraction*
$x := f(x_1, \cdots, x_n)$	$\hat{x} = \widehat{x_1} = \cdots = \widehat{x_n}$
$x := x_1 \ \$\ init\ c$	$\hat{x} = \widehat{x_1}$
$x := x_1\ when\ x_2$	$\hat{x} = \widehat{x_1} \wedge \widehat{t\_\_x_2}$ $\widehat{t\_\_x_2} = \widehat{x_2} \wedge x_2$
$x := x_1\ default\ x_2$	$\hat{x} = \widehat{x_1} \vee \widehat{x_2}$ $\widehat{def\_\_x_2} = \widehat{x_2} \backslash \widehat{x_1}$

[6] An expression is said to be defined if all the variables it contains are present.

S-CGA expresses the relations between signals, including the relations between clocks and the relations between values, which are consistent with the SIGNAL program. Thus, in the OCaml code, the normalized program is transformed into S-CGA by using two functions: *trans_eqns_clkeqv* and *attachEquation*.

The function *trans_eqns_clkeqv* is used to construct the clock equations from S-CGA. *KAss((s, k),ex)* considers the three cases in the instantaneous function $x := f(x_1, \cdots, x_n)$. The constructor *KAssSig(s,es)* deals with the primitive constructs *delay*, *undersampling* and *deterministic merging*. Here, all the primitive constructs have been flattened.

```
let  trans_eqns_clkeq eqs =
 uniq (List.concat (List.map (
  function
  |KAss((s,k), ex) ->
    begin match ex with
     SVar(s1,Ksignal) ->
      [Dass(SVar("true",Ksignal),
       (s,Clock), SVar(s1,Clock))]
     |SFunc (f, [(s1,k)]) ->
      [Dass(SVar("true",Ksignal),
       (s,Clock), SVar(s1,Clock))]
     |SFunc (f, [(s1,k1);(s2,k2)]) ->
      [Dass(SVar("true",Ksignal),
      (s,Clock), SVar(s1,Clock));
       Dass(SVar("true",Ksignal),
       (s,Clock), SVar(s2,Clock))]
     |_ -> [] (*in case of constant*)
        end
  |KAssSig(s, es) ->
    begin match es with
     KDelay(s1,e) ->
      [Dass(SVar("true",Ksignal),
       (s,Clock), SVar(s1,Clock))]
     |KWhen(s1,s2) ->
      [Dass(SVar("true",Ksignal),
      ("t___" ^ s2,Clock),
       SFunc("and", [(s2,Clock);(s2,Ksignal)]));
       Dass(SVar("true",Ksignal),(s,Clock),
       SFunc("and", [(s1,Clock);
       ("t___" ^ s2,Clock)]))]
     |KDefault(s1,s2) ->
      [Dass(SVar("true",Ksignal),(s,Clock),
       SFunc("or", [(s1,Clock);(s2,Clock)]));
       Dass(SVar("true",Ksignal),
       ("def___" ^ s2,Clock),
      SFunc("diff", [(s2,Clock);(s1,Clock)]))]
     end
   ) eqs))
```

Moreover, the function *attachEquation* is used to construct the relation between values from S-CGA.

```
let attachEquation eqns=
 List.iter(
 function
  |KAssSig(s, es) ->
   begin match es with
  |KDelay(s1, se1) ->
    Hashtbl.add dces (SVar(s,Clock))
    (KAss((s,Ksignal),se1));
    (*init(^y)=>y=d*)
    Hashtbl.add ddes (SVar(s,Clock))
    (KAss((s,Ksignal),SVar(s1,Ksignal)))
    (*^y=>next(y)=x*)
  |KDefault(se1, se2) ->
     Hashtbl.add clseq (SVar(se1,Clock))
     (KAss((s,Ksignal),SVar(se1,Ksignal)));
   Hashtbl.add clseq(SVar("def___" ^ se2,Clock))
     (KAss((s,Ksignal),SVar(se2,Ksignal)))
  |KWhen (se1, se2) ->
     Hashtbl.add clseq (SVar(s,Clock))
     (KAss((s,Ksignal), SVar(se1,Ksignal)))
   end
  |KAss((s,k), ex) as e ->
     Hashtbl.add clseq (SVar(s,Clock)) e;
  ) eqns;
  clseq
```

4.4 Solving the clock equation system

We first introduce the resolution algorithm, then we present how to compute clock equivalence classes to avoid multiple definitions of a clock, and finally get a reduced clock equation system (namely reduced normal form) based on the clock equivalence classes.

(1) Transformation to normal form

The clock equation system extracted from the S-CGA program will be solved as a system of equations of the form $h = h_1 \langle op \rangle h_2$ (which is also called *normal form*), to efficiently check the presence of a clock. That's why, for example, if the equation system contains more than one equation with the same clock on the left side, the execution of the generated code will check the same control condition several times, which is inefficient. As shown in section 3.2, there are also other problems that need to be dealt with, such as *clock-to-clock cycles* and *complex relations*.

Given two sets *NFS* and *UNFS*, *NFS* is a set of normalized clock equations, and *UNFS* is a set of unnormalized ones for example *complex relations*. In *NFS*, each clock equation is uniquely defined to avoid *multiple definitions*, and the clock variable in the left-hand side (LHS) cannot be in the right-hand side (RHS) to avoid *clock-to-clock cycles*.

Algorithm 1: Resolution of the clock equation system.

- Step 1: given any equation *eq* of the clock equation system, replace the clock variables in both sides of *eq* with the corresponding definition which has been defined in

NFS (if it has been defined), and we get the new equation *eq'*.
- Step 2: if the LHS of *eq'* is a clock expression and its RHS is a clock variable, then we reverse *eq'*.
- Step 3: if the LHS of *eq'* is a clock variable, and it exists in the RHS, then Step 5, else Step 6.
- Step 4: if both LHS and RHS of *eq'* are clock expressions, namely a complex relation, then Step 5.
- Step 5: put *eq'* into *UNFS*, then Step 8.
- Step 6: put *eq'* into *NFS*, then Step 7.
- Step 7: for each equation *ceq* in *UNFS*, replace the clock variables of both sides of *ceq* with *eq'*, and we get a new equation *ceq'*, if the LHS and the RHS of *ceq'* are equivalent, then we eliminate it from *UNFS*, else we add it into *UNFS*.
- Step 8: repeating Step 1 - Step 7 until there is no equation in the clock equations system.
- Step 9: if *UNFS* is empty, then return *NFS*, else return a refuse information.

The corresponding OCaml expression is given as follows. Here, we have two lists *UNFS* and *NFS*. When we add a new equation to the list *NFS*, we need to replace other equations in *NFS* using the new one.

```
let cleqs2nf actionlist =
  let cleqslist= action2cleqslist actionlist in
  let nfeqnlist= seqn2nfeqnlist cleqslist in
  let (unfs,nfs)=
  List.fold_right(
    fun(NFAss (cl,clexp))(unfs, nfs)->
    let l=nfsreplace nfs (clk2exp cl)
    and r= nfsreplace nfs clexp in
    if basic l then
    if occurs l r then
    (add_unfs l r unfs, nfs)
    else
    (apply(NFAss(exp2clk l, r)) unfs,
     add_nfs l r nfs)
    else if basic r then
    if occurs r l then
    (add_unfs r l unfs, nfs)
    else
    (apply(NFAss(exp2clk r, l)) unfs,
     add_nfs r l nfs)
    else
    (add_unfs l r unfs, nfs)
  ) nfeqnlist ([],[])
  in
  if unfs=[] then
   nfs
  else
   failwith("cleqs2nf␣cannot␣solve␣the␣system")
```

In the SIGNAL compiler, the clock analysis mainly relies on a Boolean abstraction of programs, internally represented as BDDs or SMTs for an efficient reasoning. In the Step 7, we reuse BDD technology to check the equivalence between LHS and RHS of a complex relation in *UNFS*.

(2) Computing clock equivalence class

As mentioned above, each clock equation is uniquely defined in *NFS* to avoid multiple definitions. Here all the clock equations are considered as boolean equations, so we reuse the BDD technology to check the equivalence of boolean equations, and put the corresponding clock variables into the same equivalence class.

The OCaml expression is given as follows. Every boolean equation has a unique BDD representation (*exp2bdd*), which corresponds to an equivalence class. Moreover, we use a hash table (*esmap*) to save the corresponding relations between equivalence class and the clock variables it contains.

```
let rec eq_class_exp=
 function
  |NFVar(sv,k1) as e ->
   let id=Clockeq2nf.exp2quotient e in
    Hashtbl.replace esmap (exp2clk e) id
  |NFFunc(f, [ce]) -> eq_class_exp ce
  |NFFunc(f, [ce1;ce2]) ->
   eq_class_exp ce1;eq_class_exp ce2
  |_ -> failwith"eq_class_exp"

let cpt_eq_class cleqslist = List.iter(
 function
  NFAss(cl,clexp) -> let id =
    Clockeq2nf.exp2quotient clexp in
     Hashtbl.replace esmap cl id ;
     eq_class_exp clexp;
 )
```

(3) Getting reduced normal form

To further efficiently check the presence of a clock, we use the identity of the equivalence class to replace the clock variables in the normal form, and we just preserve one equation for one identity. This form is called *reduced normal form*. Here, we give the abstract syntax of the reduced normal form in OCaml.

```
type classid=
        Classid of UseBDD.BDD.t
type classexpr=
        CClassid of UseBDD.BDD.t
        |Notcl of classexpr
        |Andcl of classexpr * classexpr
        |Orcl of classexpr * classexpr
        |Diffcl of classexpr * classexpr
type classeqs = (classid, classexpr) Hashtbl.t
```

4.5 Hierarchical representation of clock equations

Before the construction of clock hierarchy, we add two steps, i.e., associating actions to the clock equivalence class and deterministic sorting.

(1) Associating actions to the clock equivalence class

The construction of clock hierarchy is based on the reduced normal form, that is each node of the clock hierarchy is an identity of a clock equivalence class. As well, we associate the actions, such as inputs, outputs and value assignments, to each clock equivalence class.

Compared with other constructs, the delay construct is a dynamic operator since the properties induced on the signals refer to different values of time indexes, so we can't associate its actions statically to the clock equivalence class (it will be dealt with in the step of code generation). As shown in section 4.3, the construct $x := x_1\ default\ x_2$ will be split as two assignments: $x = x_1$ and $x = x_2$. The construct $x := x_1$ *when* x_2 is translated into a unique assignment $x = x_1$ in the equivalence class of $\hat{x}$.

The OCaml expression is given as follows.

```
let clseq=Hashtbl.create 1000
let ddes=Hashtbl.create 1000
let dces=Hashtbl.create 1000
let attachEquation eqns=
List.iter(
 function
   | SAssSig(s, es) ->
     begin match es with
     | SDelay(s1, se1) ->
       Hashtbl.add ddes (SAssSig(s,es))
                        (SAssSig(s,es))
     | SDefault (se1, se2) ->
       let id1=Hashtbl.find(Cpt_eqclass.esmap)
                           (Hat(se1)) in
       let id2=Hashtbl.find (Cpt_eqclass.esmap)
               (Hat("def___" ^ se2)) in
       Hashtbl.add clseq id1 (SAss(s,SIdent(se1)));
       Hashtbl.add clseq id2 (SAss(s,SIdent(se2)));
       Hashtbl.add dces (SAss(s,SIdent(se1)))
                        (SAss(s,SIdent(se1)));
       Hashtbl.add dces (SAss(s,SIdent(se2)))
                        (SAss(s,SIdent(se2)))
     | SWhen (se1, se2) ->
       let id1=Hashtbl.find(Cpt_eqclass.esmap)
                           (Hat(s)) in
       Hashtbl.add clseq id1(SAss(s, SIdent(se1)));
       Hashtbl.add dces (SAss(s,SIdent(se1)))
                        (SAss(s,SIdent(se1)))
      end
   | SAss(s, ex) as e ->
       let id1=Hashtbl.find (Cpt_eqclass.esmap)
                            (Hat(s)) in
       Hashtbl.add clseq id1 e;
       Hashtbl.add dces e e
         ) eqns;
         clseq
let pl_attacheq pl=List.map(
   function
     SProc(n,is,os,eqns,ws) ->
         input:=!input@is;
         output:=!output@os;
         let cls=attachEquation eqns in
         cls
   ) pl
```

(2) Deterministic sorting

The code generation is based on both the clock hierarchy and the conditional data dependencies. However, there may be clock-to-data cycles when we combine the two intermediate representations. To cut the cycles, we first sort the clock relations defined by the reduced normal form and the data dependencies represented by the actions (such as inputs, outputs, and value assignments) which have been associated to the clock equivalence classes, then we can get a total order for the clock hierarchy because of the *endochrony* property.

The OCaml expression is given as follows. We use three values -1, 0, 1 to express the order between two equations for example eq_1 and eq_2, if eq_1 is executed before eq_2, then it returns -1, if eq_1 is executed after eq_2, then it returns 1, else it returns 0. The sorting principle is defined as, for example, if the LHS of eq_1 exists in the RHS of eq_2, then eq_1 is executed after eq_2, if the LHS of eq_2 exists in the RHS of eq_1, then eq_1 is executed before eq_2.

```
type saction=
    |Dass of string * sexp
    |Cass of classid * classexpr
    |Input of string
    |Output of string
let act_comp act1 act2=
    if act1=act2 then 0
    else
     match (act1, act2) with
     |(Dass(s1,e1), Dass(s2,e2)) ->
       if var_in_sexp s1 e2 then -1
       else if var_in_sexp s2 e1 then 1
       else compare act1 act2
     |(Dass(s1,e1), Cass(Classid cl2,cle2)) ->
       if var_in_classexpr s1 cle2 then -1
       else if sexp_in_classid cl2 e1 then 1
       else compare act1 act2
     |(Cass(cl1,cle1), Cass(cl2,cle2)) ->
       if classid_in_classexpr cl1 cle2 then -1
       else if classid_in_classexpr cl2 cle1 then 1
       else compare act1 act2
     |(Cass(Classid cl1,cle1), Dass(s2,e2)) ->
       if sexp_in_classid cl1 e2 then -1
       else if var_in_classexpr s2 cle1 then 1
       else compare act1 act2
```

```
|(Input s1, Input s2) -> compare s1 s2
|(Input s1, _) -> -1
|(_, Input s2) -> 1
|(Output s1, Output s2) -> compare s1 s2
|(Output s1, _) -> 1
|(_, Output s2) -> -1
```

(3) Construction of a clock hierarchy

It will be easier to construct a clock hierarchy based on the deterministic sorting. Here we consider the sorting as a depth first search (DFS) order. However, the resulting tree-based representation of the equation systems over clocks can be optimized in the sense that the insertion step during the fusion chooses a parent with greatest depth, i.e., insertion as deep as possible.

As shown in Fig.2, according to the algorithm [36] that has been implemented in the compiler, *C5* will be inserted at the right side of *C4*, *C2*, *C4* and *C5* have the same parent node *C*. If there exists clock inclusion $C5 \Rightarrow C4$, we can insert *C5* as a son node of *C4* and at the right side of *C42*. As shown in section 4.3, we have considered the condition-clocks ($[x]$, $[\neg x]$) as white boxes, so here we can get more precise clock inclusion relations. Moreover, when we insert *C5*, all the actions which are executed before *C5* have been inserted into the hierarchy, so there is a *limit branch* of which we can just insert the new node into the right side. To insert as deep as possible, we can check the clock inclusion relations between *C5* and all the nodes which are executed after this limit branch.

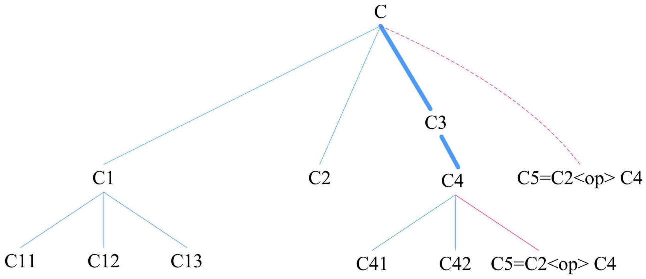

Figure 2 Optimization of the clock hierarchy

We give the basic algorithm as follows.

Algorithm 2: Construction of a clock hierarchy.

- Step 1: get an element from the sorted list.
- Step 2: if the current element is an equation from reduced normal form, namely it is the relation between clock equivalence classes, we find its limit branch on the hierarchy, and find a good insertion place based on clock inclusions, then create a new node for the equivalence class of the LHS of the equation.
- Step 3: if the current element is an action such as an input, an output, or a value assignment, we find the corresponding equivalence class which has been inserted in the hierarchy, and insert this action to the action list of the corresponding equivalence class (here we don't need to sort the action list, because they have been sorted in the sorting list).
- Step 4: Repeating Step 1- Step 3, until there is no element in the sorting list.

4.6 Code generation and optimization

Here, the basic idea of code generation (Fig.3) is as in the SIGNAL compiler: it is strongly guided by the clock hierarchy resulting from the clock calculus to structure the target language program, and by the data dependencies not only to locally order elementary operations in sequences, but also to schedule component activations in a hierarchical target code. However, all the actions have been added in the clock hierarchy, so the code generation become simpler. Moreover, the generated code can be more optimized because of previous enhancements. Furthermore, we can also do some optimizations based on clock inclusions at the generated code level.

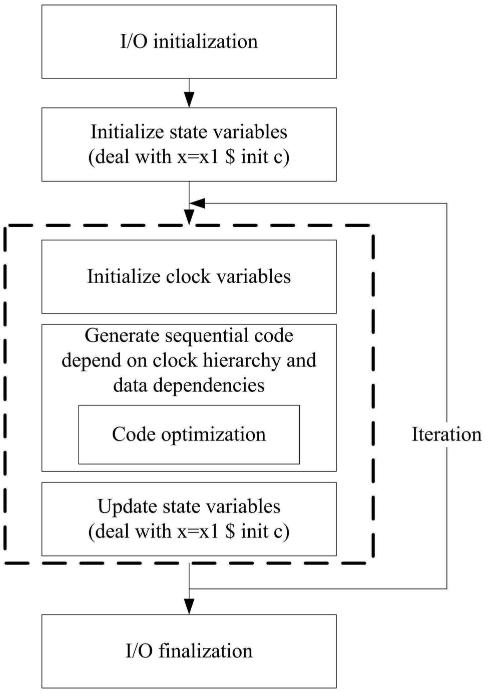
I/O initialization
Initialize state variables
(deal with x=x1 $ init c)
Initialize clock variables
Generate sequential code
depend on clock hierarchy and
data dependencies
Code optimization
Update state variables
(deal with x=x1 $ init c)
Iteration
I/O finalization

Figure 3 Sequential code generation from SIGNAL programs

5 Related Work

We discuss in this section some related studies about two aspects: enhancements of the SIGNAL compilation process and validation of the SIGNAL compilation process.

5.1 Enhancements of the SIGNAL compilation process

In the SIGNAL compiler, the control flow expressed by abstract clocks serves to derive a control structure in automatic code generation. Thus, the quality of clock calculus has a strong impact on the correctness and efficiency of implementations. There is some research about the enhancement of the clock calculus of synchronous languages.

In [10], the authors denote that there is a limitation of the clock calculus of the SIGNAL compiler, that is the SIGNAL compiler does not fully handle numerical expressions. This has a strong impact on the analysis precision and on the quality of generated code. Thus, the authors propose a new clock abstraction, that is *combined numerical-Boolean* abstraction, to eliminate this problem. In the new abstraction, every signal in a program is associated with a pair of the form (*clock*, *value*), where clock is a Boolean function and value is a Boolean or numeric function. They also use a SMT (Satisfiability Modulo Theory) solver to reason on the new abstraction.

With the same purpose, in [11] and [12], an interval-based data structure referred to as Interval-Decision Diagram (IDD) is considered for the analysis of numerical properties in SIGNAL programs.

In [37], the authors address the static analysis and code generation for applications defined in MRICDF (Multi-Rate Instantaneous Channel Connected Data Flow), which is a visual actor-oriented polychronous formalism, strongly inspired by SIGNAL. The static analysis in MRICDF also relies on a Boolean encoding of specifications, thus ignoring non-Boolean properties. Moreover, a SMT-based implementation of this static analysis is proposed as an efficient alternative to the initial implementation using a prime implicant generator.

5.2 Validation of the SIGNAL compilation process

For a safety-critical system, it is naturally required that the compiler must be formally verified as well to ensure that the source program semantics is preserved. There are many approaches to gain assurance that the transformation or the translation of a specification or a program is semantic-preserving. This can be done by directly building a theorem-prover-verified compiler [38, 39], by using translation validation [32, 40], proof-carrying code [41], semantics-anchoring method [42, 43], etc.

A. Pnueli et al. propose the approach of translation validation to verify the code generator of SIGNAL [32]. In that work, the authors define a language of symbolic models to represent both the source and target programs, called Synchronous Transition Systems (STS). A STS is a set of logic formulas which describe the functional and temporal constraints of the whole program and its generated code. Then they use BDD (Binary Decision Diagrams) representations to implement the symbolic STS models, and their proof method uses a SAT-solver to reason on the signal constraints. The drawback of this approach is that it does not capture explicitly the clock semantics and in some cases, the code generator eliminates the use of a local register variable in the generated code and then, the mapping cannot be established. Additionally, for a large program, the formula is very large, including numerical expressions that make some inefficiency. Moreover, the whole calculation of a synchronous program or the generated code is considered as one atomic transition in STS, thus it does not capture the data dependencies between signals and does not explicitly prove the preservation of abstract clocks in the compiler transformations.

In [44], the authors adopt translation validation to formally verify that the clock semantics and data dependence are preserved during the compilation of the SIGNAL compiler. They represent the clock semantics, the data dependence of a program and its transformed counterpart as first-order formulas which are called Clock Models and Synchronous Dependence Graphs (SDGs) respectively. Then they introduce clock refinement and dependence refinement relations which express the preservations of clock semantics and dependence, as a relation on clock models and SDGs respectively. Finally, a SMT-solver is used for checking the existence of the correct transformation relations.

In the work of [45], the authors encode the source SIGNAL programs and their transformations with Polynomial Dynamical Systems (PDSs), and prove that the transformations preserve the abstract clocks and clock relations of the source programs. By using simulation based on model checking techniques, this approach suffers from the increasing of the state-space when it deals with large programs.

These existing research mainly use the method of translation validation. However, translation validation treats the compiler as a "black box", namely it checks the input and output of each program transformation to validate the semantics preservation. So it yields that one need to redo the validation when the source program is changed.

In this paper, the OCaml implementation is a basis for the formal verification of the compilation of SIGNAL with a theorem prover. This work would be reused by ones interested in experimenting a new strategy for clock calculus and experimenting a new proof technique for the correctness of clock calculus. Actually, in [21], we have started the proof of semantics preservation of the front-end of our compiler, which is mechanized in the theorem prover Coq.

6 Conclusion and Future Work

This paper presents a simple and safe compiler, called MinSIGNAL, from a subset of the synchronous dataflow language SIGNAL to C, as well as its existing enhancements. For the community of statically typed functional languages, usual arguments on quality, safety and efficiency about code written in OCaml are well known and accepted for a long time. Thus, OCaml is used for the implementation of MinSIGNAL. An analysis of the SIGNAL compilation process and the existing enhancements are first given. Then, the compiler MinSIGNAL is presented step by step in details. This work is a basis for the formal verification of the compilation of SIGNAL with a theorem prover such as Coq.

In [21], we have started the proof of semantics preservation of the front-end of our compiler, which is mechanized in the theorem prover Coq. Moreover, we have given the back-end of the compiler, including sequential code generation and multi-threaded code generation with time-predictable properties. With the rising importance of multi-core processors in safety-critical embedded systems or cyber-physical systems (CPS), there is a growing need for model-driven generation of multi-threaded code and thus mapping on multi-core. We are currently working on the proof of semantics preservation of the back-end of our compiler.

Acknowledgements This work is supported in part by the National Natural Science Foundation of China under Grant 61502231, the National Defense Basic Scientific Research Project under Grant JCKY2016203B011, the Natural Science Foundation of Jiangsu Province under Grant BK20150753, the Project of the State Key Laboratory of Software Development Environment of China under Grants SKLSDE-2015KF-04, the Avionics Science Foundation of China under Grant 2015ZC52027, China Postdoctoral Science Foundation, and the Open Project of Shanghai Key Laboratory of Trustworthy Computing under Grant 07dz22304201502, and the RTRA STAE Foundation in France (http://www.fondation-stae.net/).

References

1. Harel D, Pnueli A. On the development of reactive systems. Logics and Models of Concurrent Systems,1989, F(13):477-498.
2. Castéran P, Bertot Y. Interactive theorem proving and program development: Coq'Art: The Calculus of inductive constructions. 2004.
3. Potop-Butucaru D, De Simone R, Talpin J P. The synchronous hypothesis and synchronous languages. The Embedded Systems Handbook, 2005, 1-21.
4. Boussinot F, De Simone R. The Esterel language. Proceedings of the IEEE, 1991, 79(9): 1293-1304.
5. Halbwachs N, Caspi P, Raymond P, Pilaud D. The synchronous dataflow programming language Lustre. Proceedings of the IEEE, 1991, 79(9):1305-1320.
6. Benveniste A, Le Guernic P, Jacquemot C. Synchronous programming with events and relations: the Signal language and its semantics. Science of Computer Programming, 1991, 16(2):103-149.
7. Schneider K. The synchronous programming language QUARTZ. Internal report, Department of Computer Science, University of Kaiserslautern, Germany, 2010.
8. Teehan P, Greenstreet M, Lemieux G. A Survey and Taxonomy of GALS Design Styles. IEEE Design and Test of Computers, 2007, 24(5):418-428.
9. Benveniste A, Caillaud B, Le Guernic P. From synchrony to asynchrony. In: Proceedings of International Conference on Concurrency Theory. 1999, 162-177.
10. Feautrier P, Gamatié A, Gonnord L. Enhancing the Compilation of Synchronous Dataflow Programs with a Combined Numerical-Boolean Abstraction. Technical Report 2nd Version, July 2013.
11. Gamatié A, Gautier T, Le Guernic P. Towards Static Analysis of SIGNAL Programs using Interval Techniques. In: Proceedings of Synchronous Languages, Applications, and Programming. 2006.
12. Gamatié A, Gautier T, Besnard L. An Interval-Based Solution for Static Analysis in the SIGNAL Language. In: Proceedings of Engineering of Computer Based Systems. 2008, 182-190.
13. Dijkstra E W. Guarded Commands, Nondeterminacy and Formal Derivation of Programs. Communications of The ACM, 1975, 18(8):453-457.
14. Brandt J, Gemunde M, Shukla S K, Talpin J P. Integrating system descriptions by clocked guarded actions. In: Proceedings of Forum on Specification and Design Languages. 2011, 1-8.
15. Brandt J, Schneider K. Separate translation of synchronous programs to guarded actions. Internal Report 382/11, Department of Computer Science, University of Kaiserslautern, 2011.
16. Brandt J, Schneider K, Shukla S K. Translating concurrent action oriented specifications to synchronous guarded actions. In: Proceedings of the ACM SIGPLAN/SIGBED 2010 conference on Languages, compilers, and tools for embedded systems. 2010, 47-56.
17. Edwards S A, Tardieu O. SHIM: a deterministic model for heterogeneous embedded systems. IEEE Transactions on Very Large Scale In-

tegration Systems, 2006, 14(8):854-867.

18. Brandt J, Gemunde M, Schneider K, Shukla A K, Talpin J P. Representation of synchronous, asynchronous, and polychronous components by clocked guarded actions. Design Automation for Embedded Systems, 2014, 18:63-97.
19. Esprit project: Safety Critical Embedded Systems SACRES. The declarative code DC+, version 1.4. Technical report, IRISA, november 1997.
20. Yang Z B, Bodeveix J P, Filali M, Hu K, Ma D F. A verified transformation: from polychronous programs to a variant of clocked guarded actions. In: Proceedings of the 17th International Workshop on Software and Compilers for Embedded Systems. ACM, 2014,128-137.
21. Yang Z B, Bodeveix J P, Filali M, Hu K, Zhao Y W, Ma D F. Towards a verified compiler prototype for the synchronous language SIGNAL. Frontiers of Computer Science, 2016, 10(1): 37-53.
22. RTCA/DO-178B Software Considerations in Airborne Systems and Equipment Certification. 1992.
23. RTCA/DO-178C Software Considerations in Airborne Systems and Equipment Certification.2011.
24. Pouzet M. Lucid Synchrone, version 3. Tutorial and reference manual. Université Paris-Sud, LRI.2016.
25. Forget J. A synchronous language for critical embedded systems with multiple real-time constraints. Ph.D. dissertation, 2009.
26. Pagano B, Andrieu O, Moniot T, Canou B, Chailloux E, Wang P, Manoury P, Colaço J L. Using Objective Caml to Develop Safety-Critical Embedded Tools in a Certification Framework. In: Proceedings of the 14th ACM SIGPLAN international conference on Functional programming, 2009, 215-220.
27. Cousot P, Cousot R, Feret J, Mauborgne L, Miné A, Monniaux D, Rival X. The ASTRÉE analyzer. In: Proceedings of the 14th European Symposium on Programming, volume 3444 of Lecture Notes in Computer Science. 2005: 21-30.
28. Besnard L, Gautier T, Le Guernic P. SIGNAL V4 Reference Manual, 2010.
29. Gamatié A. Designing embedded systems with the Signal programming language: synchronous, reactive specification. Springer Science and Business Media, 2009.
30. Le Guernic P, Gautier T. Data-Flow to von Neumann: the Signal approach. Advanced Topics in Data-Flow Computing, 1991, 413-438.
31. Le Guernic P, Talpin J P, Le Lann J C. Polychrony for system design. Journal of Circuits, Systems, and Computers, 2003, 12(03): 261-303.
32. Pnueli A, Siegel M, Singerman E. Translation validation. In: Proceedings of International Conference on Tools and Algorithms for the Construction and Analysis of Systems. Springer Berlin Heidelberg, 1998, 151-166.
33. Talpin J P, Brandt J, Gemunde M, Schneider K, Shukla S K. Constructive polychronous systems. In: Proceedings of International Symposium on Logical Foundations of Computer Science. Springer Berlin Heidelberg, 2013, 335-349.
34. Brandt J, Gemunde M, Schneider K, Shukla S K, Talpin J P. Embedding polychrony into synchrony. IEEE Transactions on Software Engineering, 2013, 39(7): 917-929.
35. Yang Z B, Bodeveix J P, Filali M. A comparative study of two formal semantics of the SIGNAL language. Frontiers of Computer Science, 2013, 7(5): 673-693.
36. Amagbegnon T, Besnard L, Le Guernic P. Arborescent canonical form of boolean expressions. Irisa, 1994.
37. Jose B A, Gamatié A, Ouy J, Shukla S K. SMT based false causal loop detection during code synthesis from polychronous specifications. In: Proceedings of 9th IEEE/ACM International Conference on Formal Methods and Models for Codesign (MEMOCODE), 2011, 109-118.
38. Leroy X. Mechanized semantics for compiler verification. In: Proceedings of International Conference on Certified Programs and Proofs. 2012, 4-6.
39. Leroy X. Formal verification of a realistic compiler. Communications of the ACM, 2009, 52(7): 107-115.
40. Necula G C. Translation validation for an optimizing compiler. ACM SIGPLAN notices. ACM, 2000, 35(5): 83-94.
41. Necula G C. Proof-carrying code. In: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. 1997, 106-119.
42. Chen K, Sztipanovits J, Abdelwalhed S, Jackson E. Semantic anchoring with model transformations. In: Proceedings of European Conference on Model Driven Architecture-Foundations and Applications. 2005, 115-129.
43. Narayanan A, Karsai G. Using semantic anchoring to verify behavior preservation in graph transformations. Electronic Communications of the EASST, 2006, 4.
44. Talpin J P, Gautier T, Le Guernic P, Besnard L. Formal verification of synchronous data-flow program transformations toward certified compilers. Frontiers of Computer Science, 2013, 7(5): 598-616.
45. Talpin J P, Gautier T, Le Guernic P, Besnard L. Formal verification of compiler transformations on polychronous equations. In: Proceedings of International Conference on Integrated Formal Methods. 2012, 113-127.

Dr. Zhibin YANG is an associate pro-fessor at Nanjing University of Aero-nautics and Astronautics, China. He received his PhD degree in Computer Science from Beihang University, Bei-jing, China in February 2012. From April 2012 to December 2014, he was a Postdoc in IRIT of University of Toulouse, France. His research interests include safety-critical real-time system, formal verification, AADL, and synchronous languages.

Dr. Jean-Paul BODEVEIX received a PhD of Computer Science from the University of Paris-Sud 11 in 1989. He has been assistant professor at University of Toulouse III since 1989 and is now Professor of computer science since 2003. His main research inter-ests concern formal specifications, automated and assisted verification of protocols as well as of proof en-vironments. He has participated in European and national projects related to these domains. His current activities are linked to real time modeling and verification either via model checking techniques or at the semantics level.

Dr. Mamoun FILALI is a full time researcher at CNRS (Centre National de la Recherche Scientifique). His main research interests concern the certified development of embedded systems. He is concerned by formal methods, model checking and theorem proving. During the last years, he has been mainly involved in the french nationwide TOPCASED project where he was concerned by the verification topic. He has also participated to the proposal of the AADL behavioral annex which has been been adopted as part of the AADL SAE standard.