Enhancing the Compilation of Synchronous Dataflow Programs with a Combined Numerical-Boolean Abstraction

Paul Feautrier, Abdoulaye Gamatié, Laure Gonnord

Research Report

Paul Feautrier, Abdoulaye Gamatié and L. Gonnord
July 2013 (V2)

LIP, University of Lyon, France

and LIRMM, CNRS, France

and LIFL, University of Lille, France

Enhancing the Compilation of Synchronous Dataflow Programs with a Combined Numerical-Boolean Abstraction

Abstract

In this research report, we propose an enhancement of the compilation of synchronous programs with a combined numerical-Boolean abstraction. While our approach applies to synchronous dataflow languages in general, here, we consider the SIGNAL language for illustration. 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. Given the performance level reached by recent progress in Satisfiability Modulo Theory (SMT), we use an SMT solver reason on this abstraction. Through sample examples, we show how our solution is used to determine absence of reaction captured by empty clocks; mutual exclusion captured by two or more clocks whose associated signals never occur at the same time; or hierarchical control of component activations via clock inclusion. We also show this analysis improves the quality of the code generated automatically by a compiler, *e.g.*, a code with smaller footprint, or a code executed more efficiently thanks to optimizations enabled by the new abstraction. The implementation of the whole approach includes a translator of synchronous programs towards the standard input format of SMT solvers, and an *ad hoc* SMT solver that integrates advanced functionalities to cope with the issues of interest in this work.

Enhancing the Compilation of Synchronous Dataflow Programs with a Combined Numerical-Boolean Abstraction

Paul Feautrier[*], Abdoulaye Gamatié[†], and Laure Gonnord [‡]

July 9, 2013

1 Introduction

Embedded systems are omnipresent in our daily life. They are typically found in consumer electronics, automotive and avionic systems, and medical systems. In most of these application domains, systems are safety-critical. They therefore call for well-suited design approaches that can fulfill their stringent requirements.

Synchronous languages [1] have been introduced in the early 80's in order to address the reliable development of safety-critical embedded systems. Some of these languages are LUSTRE [2], ESTEREL [3] and SIGNAL [4]. Nowadays, they are successfully adopted by the European industry as illustrated by the use of the Scade tool to develop the Airbus A380 control and display system. Among the features that make synchronous programming suitable for the design of safety-critical systems, we mention their mathematical foundation that offers a precise semantics of programs, a trustworthy reasoning on program properties, and automatic generation of correct-by-construction implementations.

Synchronous languages consider a high abstraction level for system design. A central assumption is that computation and communications are instantaneous from the viewpoint of a logical time, referred to as "*synchrony hypothesis*". This favors deterministic models of system behaviors for safe analysis. The existing synchronous languages distinguish themselves from each other by adopting different programming styles, e.g., ESTEREL has an imperative style suitable for control-dominant applications while LUSTRE and SIGNAL[1] respectively borrow functional and relational styles suitable for dataflow-oriented applications. In this paper, we mainly concentrate on the last family of languages, *i.e.*, dataflow synchronous languages.

The design approach of an embedded system with the LUSTRE language usually assumes a *reference clock* providing the time scale for all system sub-parts. In terms of set of instants, the activation clocks of sub-parts are subsets of this reference clock. While this "synchronized" model of a system is suitable for guaranteeing determinism, it suggests a monolithic vision of design so that one cannot focus on the activity of a given sub-part of a system regardless of the reference clock.

The design model adopted in the SIGNAL language is different from that of LUSTRE: the description of system sub-parts is enabled, without assuming any reference clock. It is referred to as *polychronous* model [4]. In this model, *abstract clocks*, consisting of discrete sets of logical instants at which events occur in system sub-parts, play a fundamental role in designs. They are used to describe all the control part: triggering of system components and interaction between different components. The control flow expressed by abstract clocks serves to derive an optimized control structure in automatic code generation. Thus, the quality of clock analysis has a strong impact on the correctness and efficiency of implementations.

[*]Paul Feautrier is with École Normale Supérieure de Lyon (LIP, INRIA, CNRS, UCBL), France.
[†]Abdoulaye Gamati is with LIRMM, CNRS/University of Montpellier, France.
[‡]Laure Gonnord is with LIFL, CNRS/University of Lille 1, France.

[1]Note however that the multi-clock design model associated with SIGNAL is also relevant for describing control aspects.

1.1 Compilation of programs: limitations

Beyond the usual syntax and type checking, the compilers of synchronous languages implement powerful static analysis and code optimization, allowing for a correct and efficient code generation.

In SIGNAL, the static analysis relies on a Boolean abstraction of programs, internally represented as *binary decision diagrams* (BDDs) [5] for an efficient reasoning [6]. However, one main limitation of this static analysis arises when the SIGNAL compiler addresses clock properties of a program defined by numerical expressions. Indeed, the adopted Boolean abstraction loses relevant information, which makes it quite inadequate for such a program. This has a strong impact on the analysis precision and the quality of generated code. Such an issue occurs when defining the activation clocks of a system as sets of events that occur when the values of some signals satisfy a numerical property. An example scenario is the activation of a (rescue) computation node in a fault-tolerant embedded system when a signal from executing nodes reaches a particular numerical value. In order to suitably address this issue, a new abstraction is required, which fully takes into account the numerical part beside the Boolean part of SIGNAL programs.

In the LUSTRE compilation [7], the same kind of Boolean abstraction is used before code generation. Thus, it suffers from the same lack of precision. Nevertheless, the static analysis of LUSTRE programs has been studied with various precise methods, for instance in [8] and more recently in [9], but the purpose was verification, and not the improvement of the compilation.

1.2 Contribution of this paper

We propose an enhancement of the compilation of synchronous dataflow programs with a combined numerical-Boolean sound abstraction. Here, this is mainly illustrated on SIGNAL programs. However, we believe the same workflow can be easily adapted to other synchronous dataflow languages, such as LUSTRE or MRICDF (Multi-Rate Instantaneous Channel connected Data Flow) [10]. Note that the current paper is an extended version of a previous one [11]. Our solution permits an analysis that significantly enhances the quality of the subsequent code generated by compilers, *e.g.*, a code with smaller footprint, a code executed more efficiently thanks to further optimizations.

The present tool is also an invaluable aid to debugging. For instance, as will be shown in Section 7 or in the discussion of the Bathtub example, we are able to statically detect empty clocks. Depending on the context, this can be interpreted as a proof of safety (an alarm will never sound), or as a bug (an operation on signals with incompatible clocks).

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. Given the performance level reached by recent progress in *Satisfiability Modulo Theory* (SMT) [12], we use an SMT solver to reason on the new abstraction. We show through a few examples, how relations between abstract clocks defined with numerical and logical expressions are adequately analyzed, to determine for instance absence of reactivity captured by empty clocks; mutual exclusion captured by two or more clocks whose associated signals never occur at the same time; or a better control of node activations via clock inclusion.

The advocated approach is depicted by Fig. 1. Given a synchronous dataflow program P, we define a corresponding abstraction, used to check the satisfiability of properties of interest, *i.e.*, those involving numerical expressions. For this purpose, we use an *ad hoc* SMT solver offering tailored functionality for an adequate usage in our approach. Once identified, all properties of interest are concretized into synchronous dataflow programs, which are later composed with the initial program P. The resulting composed program is equivalent to P in which properties involving numerical expressions have been made explicit in a form that is suitably addressable by a synchronous language compiler. Then, it becomes easier for the compiler to do an efficient analysis and code generation. The main part of our contribution is on the right-hand side of Fig. 1. Notice that an important advantage of this contribution is its *modular*, *i.e.*, non-intrusive, implementation regarding compilers. This clearly facilitates its integration to a given compiler and makes it easy to isolate a bug in the global framework (in comparison to a compiler-intrusive solution).

Compared to our preliminary publication [11], the present article brings new improvements re-

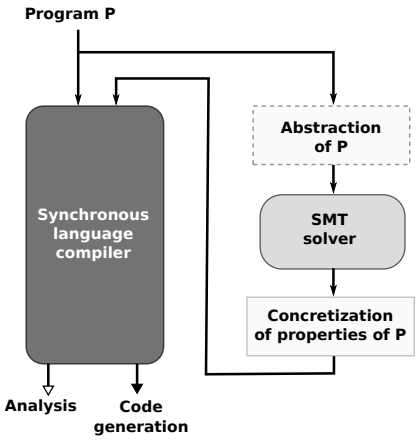

Figure 1: Overview of the proposed approach.

garding the following aspects:

- definition of an "ad hoc" SMT solver, while the off-the-shelf Yices solver was considered previously. This enables us to implement property search inside the new solver, thus avoiding costly pretty printing and parsing;
- a proposal within the same solver to compute strongly connected components of the clock implication graph for determining an enhanced clock hierachy useful to efficient code generation;
- additional examples illustrating the relevance of our solution.

1.3 Outline

The remainder of this paper is organized as follows. Section 2 compares the proposed approach to some relevant existing works. Section 3 gives an overview of SIGNAL. Section 4 discusses the current limitations of the static analysis achieved by the SIGNAL compiler, regarding clock analysis and code generation. Section 5 exposes a new combined numerical-Boolean abstraction for improving this static analysis by using first-order logic formulas. Section 6 presents an implementation of our approach. Section 7 addresses typical application examples for which our proposal is very useful. Finally, Section 8 gives concluding remarks.

2 Related work

We discuss in this section some relevant studies about static analysis techniques for synchronous programming. Since these techniques apply both to verification and compilation, we distinguish them w.r.t. both topics.

2.1 Static analysis for verification

A few combinations of numerical and Boolean verification techniques have been studied for LUSTRE verification. In [8], the technique used is a dynamic partitioning of the control flow obtained by LUSTRE compilation with respect to constraints coming from a given proof goal. Our approach does not depend on any proof goal. A recent work [13] proposed a method based on a combination of abstract acceleration techniques [14] and control-flow refinement [8] in order to prove reachability. The results are very accurate, but the analysis is very expensive to be integrated to a compiler for the moment. Our analysis is cheaper and does not suffer from the same state explosion problem.

An important work is the polyhedral-based static analysis for synchronous languages, and in particular, for the SIGNAL language [15]. The authors give a technique based on fix-point iteration on a lattice combining Boolean and affine constraints. More recently, a polyhedral analysis library has been integrated to the SIGNAL open-source compiler in order to compute safe operating ranges

for input variables of programs [16]. This was intended for an improvement of the causality analysis of SIGNAL programs. Our technique is less precise than [15] and [16] because it cannot deal with polyhedral invariants. But, the complexity of the analysis in our case is lesser and the implementation is much simpler.

In another study, a clock language $\mathcal{CL}$ has been introduced to capture the static control part of SIGNAL programs [17]. The author also considers SAT decision procedures to prove clock properties. However, statements involving the *delay* construct are not taken into account in this study. This reduces the scope of the proposed analysis. Our proposition aims to cover programs containing any construct of the SIGNAL language. In particular, regarding the *delay* construct, we propose here two abstractions with different precision levels: one solution that only captures the synchronization property related to manipulated variables (note that this property can be also addressed with $\mathcal{CL}$ even though not considered by its author); and a more precise solution that refines the first one with additional constraints on data values carried by manipulated variables. Thus, our approach slightly offers more expressivity than $\mathcal{CL}$. In addition, while the main motivation of the abstraction considered for $\mathcal{CL}$ is to prove clock properties of a subset of SIGNAL, the goal of our approach goes beyond that by focusing more generally on SIGNAL program compilation, including both clock property analysis and code generation optimization. Furthermore, compared to [17] that considers SAT solvers, here the use of SMT solvers provides a more powerful analysis, especially on numerical properties.

Finally, SMT techniques were used to verify safety properties in LUSTRE [18]. The authors consider a specific form of LUSTRE language and propose a modeling in a typed first order logic with uninterpreted function symbols and built-in integers and rationals. While this work also aims at benefiting from SMT solving in synchronous programming, it misses all useful clock analysis achieved by the SIGNAL compiler in our case. Such an analysis includes suitable heuristics to address polychronous specifications. Neither an SMT solver nor the LUSTRE compiler makes this analysis possible.

2.2 Static analysis for compilation

In [19, 20], an interval-based data structure referred to as *interval-decision diagram* (IDD) is considered for the analysis of numerical properties in SIGNAL programs. While the main idea is similar to that of this paper, the choice of SMT solvers appears however more judicious. First, in IDDs, intervals are only defined on integers. As a result, to deal with other numerical types such as reals, IDDs require a prior encoding into integers. With SMT solvers, a wide range of arithmetic theories are made possible, which allows a more expressive analysis without much effort compared to IDDs. Second, from a practical point of view, the integration of IDDs in the SIGNAL compiler is more difficult since it requires a very careful coupling with the other data structures used during the static analysis. One important question is how to make efficient and costless the management of binary decision diagrams (BDDs), which are part of IDDs and are already present in the compiler. In this paper, we rather consider the modular solution shown in Fig. 1.

The optimization of synchronous programs described as synchronous guarded actions is studied in [21]. From such descriptions, extended finite state machines (EFSMs) are generated, in which each state is associated with dataflow guarded actions to be executed in this state. EFSMs make explicit the control-flow of the sequential code to be generated from input synchronous programs (while the dataflow part is captured symbolically). Based on EFSMs, authors use an SMT solver to check the validity of guards. Valid guards lead to actions that are executed every time, while invalid guards refer to actions that are never executed, i.e., dead code. Our solution is similar to this approach. However, the abstraction we consider for SMT reasoning covers both the control part, *i.e.*, clocks, and the data part, *i.e.*, values.

Finally, in [22, 23], authors address the static analysis and code generation for applications defined in MRICDF, 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. In [22, 23], an SMT-based implementation of this static analysis is proposed as an efficient alternative to the initial implementation using a prime implicant generator. This implementation showed a noticeable speed-up. The combined numerical-Boolean abstraction proposed in the current paper can be seen as one major improvement applicable to this SMT-based implementation,

Table 1: Trace semantics for SIGNAL primitives.

process P	semantics of P: $[\![\mathtt{P}]\!]$
`y:= R(x1,...,xn)`	$\{\ T \in \mathcal{T}^{\perp}_{\{x_1,\ldots,x_n,y\}} / \ \forall t \in \mathbb{N}, \big(\forall i, T(t)(x_i) = T(t)(y) = \perp\ \big)$ *or* $\big(T(t)(y) \neq \perp$ *and* $\forall i, T(t)(x_i) \neq \perp$ *and* $T(t)(y) = R(T(t)(x_1), \ldots, T(t)(x_n))\big)\}$
`y:= x $ 1 init c`	$\{\ T \in \mathcal{T}^{\perp}_{\{x,y\}} / \ \forall t \in \mathbb{N}, \big(T(t)(x) = T(t)(y) = \perp\ \big)$ *or* $\Big(T(t)(y) \neq \perp$ *and* $T(t)(x) \neq \perp$ *and* $T(t_0)(y) = c$ *and* $\big((t \geq t_0) \Rightarrow (\exists i, t = t_i, T(t_{i+1})(y) = T(t)(x))\big)\Big)$ *with* $t_0 = \inf\{t / T(t)(x) \neq \perp\}$ and $t_{i+1} = \inf\{t / t > t_i \wedge T(t)(x) \neq \perp\}$ $\}$
`y:= x when b`	$\{\ T \in \mathcal{T}^{\perp}_{\{x,b,y\}} / \ \forall t \in \mathbb{N}, \big(T(t)(b) = true$ *and* $T(t)(y) = T(t)(x)\big)$ *or* $\big(T(t)(b) \neq true$ *and* $T(t)(y) = \perp\ \big)\}$
`z:= x default y`	$\{\ T \in \mathcal{T}^{\perp}_{\{x,y,z\}} / \ \forall t \in \mathbb{N}, \big(T(t)(x) \neq \perp$ *and* $T(t)(z) = T(t)(x)\big)$ *or* $\big(T(t)(x) = \perp$ *and* $T(t)(z) = T(t)(y)\big)\}$
$\mathtt{P}_1 \mid \mathtt{P}_2$	Assuming that $[\![\mathtt{P}_1]\!] \subseteq \mathcal{T}^{\perp}_{X1}, [\![\mathtt{P}_2]\!] \subseteq \mathcal{T}^{\perp}_{X2}$, $\{\ T \in \mathcal{T}^{\perp}_{X1 \cup X2} / \ \ X1.T \in [\![\mathtt{P}_1]\!]$ *and* $X2.T \in [\![\mathtt{P}_2]\!]\}$
$\mathtt{P}_1$ `where x`	Assuming that $[\![\mathtt{P}_1]\!] \subseteq \mathcal{T}^{\perp}_{X1}$, $\{T \in \mathcal{T}^{\perp}_{X1-\{x\}} / \exists\, T1 \in [\![\mathtt{P}_1]\!],\ (X1 - \{x\}).T1 = T\}$

as for SIGNAL.

3 Overview of the SIGNAL language

SIGNAL [4] [24] is a data-flow relational language that handles unbounded series of typed values $(x_t)_{t \in \mathbb{N}}$, called *signals*, implicitly indexed by discrete time, and denoted as `x`. For instance, a signal can be either of *Boolean* or *integer* or *real* types. At any logical instant $t \in \mathbb{N}$, a signal may be present, at which point it holds a value; or absent and denoted by $\perp$ in the semantic notation. There is a particular type of signal called `event`. A signal of this type always holds the value *true* when it is present. The set of instants at which a signal `x` is present is referred to as its *clock*, noted `^x`. A *process* is a system of equations over signals, specifying relations between values and clocks of the signals. A *program* is a process. Before presenting the primitive statements (or constructs) of SIGNAL, we introduce a denotational semantic model used to formally define these statements.

3.1 A trace denotational semantic model

We present the basic elements of a *trace semantics* [25] for SIGNAL. Let us consider a finite set $X = \{x_1, \ldots, x_n\}$ of typed variables called *ports*. For each $x_i \in X$, $\mathcal{D}_{x_i}$ is its domain of values. In addition, we have:

$$\mathcal{D} = \bigcup_{i=1}^{n} \mathcal{D}_{x_i} \text{ and } \mathcal{D}^{\perp} = \mathcal{D} \cup \{\perp\},$$

where $\perp \notin \mathcal{D}$ denotes the absence of value associated with a port at a given instant. The domains $\mathcal{D}^{\perp}_{x_i}$ and $\mathcal{D}^{\perp}_{X_1}$ are defined in a similar way with $X_1 \subseteq X$.

Definition 1 (events) *Given a non-empty set $X_1 \subseteq X$, the set of events on X_1, denoted by $\mathcal{E}_{X1}$, is the set of all applications (functions) m defined from X_1 to $\mathcal{D}^{\perp}_{X_1}$.* □

The expression $m(x) = \perp$ means x holds no value while $m(x) = v$ means that x holds the value v, and $m(X_1) = \{m(x) / x \in X_1\}$. The set of events on X_1 is denoted by $\mathcal{E}_{X_1} = X_1 \to \mathcal{D}^{\perp}_{X_1}$, and the set of all possible events is therefore $\mathcal{E} = \bigcup_{X_1 \subseteq X} \mathcal{E}_{X_1}$. By convention, the event on an empty set of ports is noted by $\mathcal{E}_{\emptyset} = \{\emptyset\}$.

Definition 2 (traces) *Given a non-empty set $X_1 \subseteq X$, the set of traces on $X1$, denoted by $\mathcal{T}^{\perp}_{X1} : \mathbb{N} \to \mathcal{E}_{X1}$, is defined by the set of applications T defined from the set $\mathbb{N}$ of natural numbers to $\mathcal{E}_{X_1}$.* □

The set of all possible traces is $\mathcal{T}^{\perp} = \bigcup_{X1 \subseteq X} \mathcal{T}^{\perp}_{X1}$. Moreover, $\mathcal{T}_{\emptyset} = \mathbf{1} = \mathbb{N} \to \mathcal{E}_{\emptyset}$.

Definition 3 (trace restriction) *Given a non-empty set $X_1 \subseteq X$, and a set $X_2 \subset X_1$ with a trace T being defined on X_1, the restriction of $T(t)$ to X_2, noted $X_2.T : \mathbb{N} \to \mathcal{E}_{X_2}$, satisfies: $\forall t \in \mathbb{N}, \forall x \in X_2 \quad X_2.T(t)(x) = T(t)(x)$.* □

We have $\emptyset.T \in \mathcal{T}_{\emptyset}$ (which is a singleton).

We extend the notion of trace restriction to a set $\mathcal{T}$ of traces on a set of variables $X \subseteq X_{\mathcal{T}}$ as follows: $X.\mathcal{T} = \{X.T | T \in \mathcal{T}\}$.

A process on a set of variables $X1 \subseteq X$ is a set of constrained traces on $X1$. In other words, it is a subset of $\mathcal{T}^{\perp}_{X1}$. The semantics of statements defining a process `P` is denoted by a set of traces $[\![\mathtt{P}]\!]$.

3.2 Primitive constructs of the language

SIGNAL relies on six primitive constructs: the *core language*. The syntax of the constructs is given below, with some informal explanations. Their formal semantics according to the trace model is summarized in Table 1.

- *Instantaneous relations:* `y:= R(x1,...,xn)` where `y, x1`, ..., `xn` are signals and `R` is a point-wise n-ary relation/function extended canonically to signals. This construct imposes `y, x1`, ..., `xn` *i)* to be simultaneously present, i.e. `^y` = `^x1` = ...= `^xn` (i.e. *synchronous* signals), and *ii)* to hold values satisfying `y = R(x1,...,xn)` whenever they occur.
- *Delay:* `y:= x $ 1 init c` where `y, x` are signals and `c` is an initialization constant. It imposes *i)* `x` and `y` to be synchronous, i.e. `^y` = `^x`, while *ii)* `y` must hold the value carried by `x` on its previous occurrence.
- *Under-sampling:* `y:= x when b` where `y, x` are signals and `b` is of Boolean type. This construct imposes *i)* `y` to be present only when `x` is present and `b` holds the value *true*, i.e. `^y` = `^x` $\cap$ $[b]$ (where $[b] \cup [\neg b]$ = `^b` and $[b] \cap [\neg b] = \emptyset$), while *ii)* `y` holds the value of `x` at those logical instants. The sub-clock $[b]$ (resp. $[\neg b]$) denotes the set of instants where `b` is *true* (resp. *false*).
- *Deterministic merging:* `z:= x default y` where `x, y` and `z` are signals. This construct imposes *i)* `z` to be present when either `x` or `y` are present, i.e. `^z` = `^x` $\cup$ `^y`, while *ii)* `z` holds the value of `x` if present, otherwise that of `y`.
- *Composition:* `P` $\equiv$ `P1|P2` where `P1` and `P2` are processes. It denotes the union of equations defined in processes, leading to the conjunction of the constraints associated with these processes. A signal variable cannot be assigned a value in `P1` and `P2` at the same time. SIGNAL adopts single assignment. A variable defined in `P1` can be an input of `P2`, and vice versa. The composition operator is commutative and associative.
- *Restriction (or Hiding):* `P` $\equiv$ `P1 where x`, where `P1` and `x` are respectively a process and a signal. It states that `x` is a local signal of process `P1`. The process `P` holds the same constraints as `P1`.

The core language of SIGNAL is expressive enough to derive new constructs of the language for programming comfort and structuring. In particular, SIGNAL allows one to explicitly manipulate clocks through some derived constructs that can be rewritten in terms of primitive ones. For instance, the clock extraction statement `y:= ^x`, meaning `y` is defined as the clock of `x`, is equivalent to `y:= (x = x)` in the core language. A similar statement `y:= when b`, defining `y` as the set of instants where the Boolean signal `b` is present and *true*, is equivalent to `y:= b when b`. The clock *union* `y:= x1 ^+ x2`, rewritten as `y:= ^x1 default ^x2`, denotes the set of instants at which at least a signal `xi` occurs. In the same way, clock *intersection* `y:= x1 ^* x2` and *difference* `y:= x1 ^- x2` are respectively defined as: `y:= ^x1 when ^x2` and `y:= when(not(^x2) default ^x1)`. The *synchronizer* `x1 ^= x2` that constrains `x1` and `x2` to have the same clock, is rewritten as `(| x:= ^x1 = ^x2 |) where x`. The *empty clock* is denoted by `^0`.

For syntactical convenience, SIGNAL enables a modular definition of processes by providing a notion of *subprocess* (or local process). The statement `P1 where P2`, where `P1` and `P2` are processes, denotes the fact that the latter process is a subprocess of the former process. Then, the body of `P1`, i.e., its associated set of equations, contains (at least) a call to process `P2`. The compilation process of SIGNAL basically inlines the body of `P2` in `P1` (with variable substitution). Note that a process `P1` may have more than one subprocess, and those subprocesses may have themselves sub-subprocesses, *ad infinitum*.

3.3 Example: a bathtub model in SIGNAL

The simple SIGNAL process shown in Fig. 2 specifies the status of a *bathtub* [15]. It has no input signal (line 02), but has three output signals (line 03).

```
-----------------------------------------------------
01:process Bathtub =
02:(?
03: ! integer level; boolean alarm, ghost_alarm; )
04:(|(| level := zlevel + faucet - pump
05:   | zlevel := level$1 init 1
06:   | faucet := zfaucet + (1 when zlevel <= 4)
07:   | zfaucet := faucet$1 init 0
08:   | pump := zpump + (1 when zlevel >= 7)
09:   | zpump := pump$1 init 0 |)
10: |(| overflow := level >= 9
11:   | scarce := 0 >= level
12:   | alarm := scarce or overflow
13:   | ghost_alarm:= (true when scarce when overflow)
14              default false |)|)
15: where
16:  integer zlevel,zfaucet,zpump,faucet,pump;
17:  boolean overflow,scarce;
18:end;
-----------------------------------------------------
```

Figure 2: A bathtub model in SIGNAL.

The signal `level`, defined at line 04, reflects the water level in the bathtub at any instant. It is determined by considering two signals, `faucet` and `pump`, which are respectively used to increase and decrease the water level. These signals are increased by one under some specific conditions (lines 06 and 08), in order to maintain the water level in a suitable range of values.

An `alarm` signal is defined at line 12 whenever the water overflows (line 10) or becomes scarce (line 11) in the bathtub. An additional "ghost" alarm is defined at line 13/14, which is not expected to occur. Here, it is just introduced to illustrate one limitation of the static analysis of SIGNAL. The clock of this signal is not completely specified in `Bathtub`. As stated in the previous section, this clock is the union of those associated with the two arguments of the `default` operator. The clock of the left argument is exactly known. The clock of the right-hand one is *contextual because the argument is a constant* (that is, a constant signal is always available whenever required by its context of usage): it is equal to the difference of `ghost_alarm`'s clock and first argument's clock. Since, this difference cannot be defined exactly from the program, further clock constraints on `ghost_alarm` will be required from the environment of `Bathtub` for an execution.

4 A limitation in the SIGNAL compiler

The static analysis of SIGNAL programs, referred to as *clock calculus*, primarily aims at proving the consistency of clock relations as well as the absence of cyclic data dependencies induced by program definition. This is necessary in order to prove the *reactivity* and the *determinism* of a modeled system. For instance, the presence of empty clocks in a program reduces its reactivity since the concerned signals are always absent. Unless such behaviors are absolutely required, they have to be avoided, in particular for the reactivity of embedded real-time systems. Determinism is characterized by the inference of a single master clock from a program. All system events are observed according to this

clock. Another property is clock *mutual exclusion*, which ensures some events never occur at the same time.

In SIGNAL, clocks are fundamentally the main means to express control (synchronizations between signals). Together with their associated relations, they are formalized through a *clock algebra* [6]. In particular, the set of clocks associated with set inclusion forms a lattice. Based on clock inclusion, the SIGNAL compiler computes a clock hierarchy on which the automatic code generation strongly relies . However, for the *under-sampling* construct, remember that the clock of the Boolean expression `b` is partitioned into $[b]$ and $[\neg b]$, which are referred to as *condition-clocks*. If `b` is defined by a numerical expression such as an integer comparison, $[b]$ and $[\neg b]$ are seen as *black boxes* when compared separately to other clock expressions. This reduces the power of the clock calculus analysis whenever a program contains numerical expressions.

4.1 Clock analysis for the bathtub model

```
-----------------------------------------------------
01:(| CLK_level := ^level
02: | CLK_level ^= alarm ^= zlevel^= faucet^= pump
02b:            ^= overflow ^= scarce
03: | CLK_zfaucet ^= when (zlevel<=4)
04: | CLK_zpump ^= when (zlevel>=7)
05: | (| CLK_level ^= CLK_zpump
06:    | CLK_level ^= CLK_zfaucet
07:    |)%**WARNING: Clocks constraints%
08: | CLK_22 := when level>=9
09: | CLK_25 := when 0>=level
10: | CLK_36 := CLK_22 ^* CLK_25
11: | (| CLK_ghost_alarm ^= CLK_36 default (not CLK_29)
12:    | CLK_29 := CLK_ghost_alarm ^- CLK_36
13:    | (| ghost_alarm := CLK_36 default (not CLK_29)
14: |) |) ... |)
-----------------------------------------------------
```

Figure 3: A sketch of clock calculus.

Fig. 3 partially shows the result of the clock calculus generated automatically by the compiler in POLYCHRONY. Here, we focus on two issues that the clock analysis was not able to fix adequately. First, a clock constraint is generated, stating that signals `CLK_level`, `CLK_zfaucet` and `CLK_zpump` must have the same clock (lines 05–07), while signals `CLK_zfaucet` and `CLK_zpump` have exclusive clocks (lines 03–04). Second, at line 11, the right-hand side of the synchronization equation about `CLK_ghost_alarm` should be `(not CLK_29)` since the clock `CLK_36` is empty by definition (line 10).

The previous two issues illustrate typical limitations of the Boolean abstraction in the clock calculus. This does not enable to verify simple static properties of a program, such as clock exclusion or emptiness, since numerical expressions are not suitably abstracted. A more expressive clock analysis would detect the fact that `CLK_level`, `CLK_zfaucet` and `CLK_zpump` must be empty clocks in order to satisfy the clock constraints of the `Bathtub` process. Section 7 discusses another issue about the hierarchical control of component activations.

4.2 Code generation of the bathtub model

The above limitations also have an important impact on the quality of the code generated automatically by the compiler since it relies on the clock hierarchy resulting from the analysis. Fig. 4 sketches a C code generated automatically based on the clock analysis.

The previous clock constraint is implemented by exception statements (lines 04–05). This can be seen currently as the way the compiler alerts a user that it was not able to solve the clock constraints related to the exception statements generated from a SIGNAL program. Of course, such a C code is only useful for simulation.

Now, if the above C code is to be embedded in some real-life system, its quality could be significantly improved by noticing that since `CLK_level`, `CLK_zfaucet` and `CLK_zpump` should be empty clocks, statements between lines 02 and 11 are never executed (and consequently, the exception statements are useless). As a result, the generated C code shown in Fig. 4 contains *dead code*. In a similar way,

```
-------------------------------------------------
01: if (C_level)
02:   { C_zfaucet = level <= 4;
03:     C_zpump = level >= 7;
04:     if ((C_zpump) != (C_level))
04b:                polychrony_exception("..." );
05:     if ((C_zfaucet) != (C_level))
05b                 polychrony_exception(" ... " );
06:     if (C_zfaucet) { faucet = zfaucet + 1; }
07:     if (C_zpump) { pump = zpump + 1; }
08:     level = (level + faucet) - pump;
09:     overflow = level >= 9; scarce = 0 >= level;
10:     alarm = scarce || overflow; ...
        /*production of level and alarm*/
11:     C_106 = overflow && scarce;} ...
12: C_109 = (C_level ? C_106 : FALSE);
13: if (C_ghost_alarm)
14:   { if (C_109) ghost_alarm = TRUE;
14b:       else ghost_alarm = FALSE;
15:     ... /* production of ghost_alarm */ } ...
-------------------------------------------------
```

Figure 4: A sketch of the generated C code.

the `if` statement at line `14/14b` also contains a dead code since the variable `ghost_alarm` is always set to *false*.

5 Our numerical-Boolean abstraction

We define an abstraction for SIGNAL program analysis. All considered programs are supposed to be in the syntax of the core language.

Our abstraction for program P is a logical formula Φ on the variables and clocks of P in a decidable theory (here, linear arithmetic of integers or reals) such that at any logical instant in an execution of P, the current values of signals and clocks satisfy Φ. In other words, at any instant in an execution of P, its variables and clocks are a model of Φ.

5.1 Notations and restrictions

Let `P` be a SIGNAL program. We denote by $X_P = \{x_1, x_2 \dots x_n\}$ the set of all variables of `P`. Here, we consider scalar variables only. With each variable x_i (numerical, Boolean or `event`), we associate two abstract values: $\widehat{x_i}$ and $\widetilde{x_i}$ encoding respectively its clock and values.

The abstract semantics of the program, is a set of couples of the form $(\widehat{\ }, \widetilde{\ })$ where:

- function $\widehat{\ }$: $X_P \to \mathbb{B} = \{true, false\}$ assigns to a variable a Boolean value;
- function $\widetilde{\ }$: $X_P \to \mathbb{R} \cup \mathbb{B}$ assigns to a variable a numerical or Boolean value.

This abstract set is represented as a first order logic formula Φ_P in which atoms are $\widetilde{x_i}$ and $\widehat{x_i}$, and the operators are usual logic operators and integer comparison functions.

5.2 Abstraction for expressions

Our abstraction strongly relies on an abstraction for expressions, detailed in the sequel.

We restrict ourselves to the following subset of numerical and Boolean expressions in SIGNAL statements. For sake of simplicity and readability, here we simplify the abstraction previously provided in [11].

$$
\begin{array}{rcl}
\mathit{nexp} & ::= & \mathit{cst} \mid \mathit{nexp} \diamondsuit \mathit{nexp} \mid \mathit{nexp} \diamondsuit' \mathit{cst} \mid \mathit{var} \\
\mathit{bexp} & ::= & \texttt{true} \mid \texttt{false} \mid \texttt{not}\ \mathit{bexp} \mid \mathit{var} \mid \mathit{bexp} \\
 & & \texttt{and}\ \mathit{bexp} \mid \mathit{bexp}\ \texttt{or}\ \mathit{bexp} \mid \mathit{nexp} \bowtie \mathit{nexp}
\end{array}
$$

where the symbols *cst* and *var* respectively denote a constant and a signal variable (`x`, `y`, ...), $\bowtie \in$ {`<`, `>`, `=`}, $\diamondsuit \in$ { `+`, `-`} and $\diamondsuit' \in$ { `/`, `*`}

The abstraction of a given numerical SIGNAL expression *nexp* (resp a Boolean expression *bexp*) will be a numerical expression (resp. a Boolean expression) that expresses its behavior.

We define an abstraction ϕ for these expressions by induction on their structure as follows:

- atoms: given a signal `x`, if `x` is of Boolean or numeric type, $\phi(\texttt{x}) = \widetilde{x}$; if `x` is of `event` type, $\phi(\texttt{x}) = true$,
- $\phi(\texttt{true}) = true$ and $\phi(\texttt{false}) = false$, and if c is a numerical constant, $\phi(c) = c$,
- if b_1 and b_2 denote Boolean expressions, then $\phi(b_1 \texttt{ and } b_2) = \phi(b_1) \wedge \phi(b_2)$; $\phi(b_1 \texttt{ or } b_2) = \phi(b_1) \vee \phi(b_2)$; $\phi(\texttt{not } b_1) = \neg\phi(b_1)$,
- if n_1 and n_2 denote numerical expressions, then $\phi(n_1 \texttt{ < } n_2) = \phi(n_1) < \phi(n_2)$, $\phi(n_1 \texttt{ > } n_2) = \phi(n_1) > \phi(n_2)$ and $\phi(n_1 \texttt{ = } n_2) = \phi(n_1) = \phi(n_2)$.
- if n_1 and n_2 denote numerical expressions, then $\phi(n_1 \texttt{ + } n_2) = \phi(n_1) + \phi(n_2)$ and $\phi(n_1 \texttt{ - } n_2) = \phi(n_1) - \phi(n_2)$
- if n is a numerical expression and c a constant, then $\phi(c \texttt{ * } n) = c.\phi(n)$ and $\phi(n \texttt{ / } c) = \frac{\phi(n)}{c}$.

The ϕ function is used to compute numerical and Boolean exact abstractions for our subset of expressions. Some approximations will be made in case of other signal expressions such as multiplication of variables, or *modulo* (an example will be found later in Section 7).

Example 1 *Let* $b = (x + y = 4)$ ***and*** $(y < 10)$ *be a Boolean expression. Its abstraction is* $\phi(b) = \widetilde{x} + \widetilde{y} = 4 \wedge \widetilde{y} < 10$.

5.3 Abstraction of SIGNAL primitive constructs

We define Φ_P as the intersection of the abstractions of statements stm_i of `P`:

$$\Phi_P = \bigwedge_i^n \Phi(stm_i)$$

where n is the number of statements composed in `P`.

Each $\Phi(stmt)$ will be a formula of quantifier-free linear integer arithmetic (QF_LIA) or quantifier-free linear real arithmetic (QF_LRA).

In the next, we distinguish two possible definitions of Φ for each primitive construct of SIGNAL, according to the type of signal `y` in each equation: (a) when `y` is of numerical type and (b) when `y` is of logical type.

- *Instantaneous relations:* `y:= R(x1,...,xn)`. The abstraction Φ of instantaneous relations is defined as follows:

$$\begin{cases} \bigwedge_{i=1}^n (\widehat{y} \Leftrightarrow \widehat{x_i}) \wedge \left(\widehat{y} \Rightarrow \widetilde{y} = \phi(nexp)\right) & (a) \\ \bigwedge_{i=1}^n (\widehat{y} \Leftrightarrow \widehat{x_i}) \wedge \left(\widehat{y} \Rightarrow (\widetilde{y} \Leftrightarrow \phi(bexp))\right) & (b) \end{cases}$$

  where `R(x1,...xn)` is denoted by either *nexp* or *bexp*.

  These expressions express the equalities between clocks and values that are induced by SIGNAL semantics.

- *Delay:* `y:= x $ 1 init c`. The abstraction Φ of the delay construct is defined as follows:

$$\widehat{y} \Leftrightarrow \widehat{x}$$

The abstraction here only expresses the equalities between clocks. A better abstraction could be performed if the user (or a pre-analysis) provides *invariants* for numerical variables. In that case, the global abstraction would be :

$$\left(\widehat{y} \Leftrightarrow \widehat{x}\right) \bigwedge \left(\widehat{y} \Rightarrow \left((invar(\widetilde{x})[\widetilde{x}/\widetilde{y}] \vee (\widetilde{y} = c)\right)\right)$$

where $invar(\widetilde{x})[\widetilde{x}/\widetilde{y}]$ denotes the substitution of $\widetilde{y}$ in a formula that expresses a constraint on x's values. Such an invariant can be a result of the methods proposed in [15] or [16].

- *Under-sampling:* `y:= x when b`. The abstraction Φ of the under-sampling construct is defined as follows:

$$\begin{cases} \left(\widehat{y} \Leftrightarrow (\widehat{x} \wedge \widehat{b} \wedge \widetilde{b})\right) \bigwedge \left(\widehat{y} \Rightarrow \widetilde{y} = \widetilde{x}\right) & (a) \\ \left(\widehat{y} \Leftrightarrow (\widehat{x} \wedge \widehat{b} \wedge \widetilde{b})\right) \bigwedge \left(\widehat{y} \Rightarrow (\widetilde{y} \Leftrightarrow \widetilde{x})\right) & (b) \end{cases}$$

which expresses the fact that the signal y is present if and only if both signals b and x are present and b is *true*. The constraints on values are straightforward.

- *Deterministic merging:* `z:= x default y`. The abstraction Φ of the deterministic merging construct is defined as follows:

$$\begin{cases} \left(\widehat{y} \Leftrightarrow (\widehat{x} \vee \widehat{z})\right) \bigwedge \\ \left(\widehat{y} \Rightarrow \left((\widehat{x} \wedge (\widetilde{y} = \widetilde{x})) \vee (\neg\widehat{x} \wedge (\widetilde{y} = \widetilde{z}))\right)\right) & (a) \\ \left(\widehat{y} \Leftrightarrow (\widehat{x} \vee \widehat{z})\right) \bigwedge \\ \left(\widehat{y} \Rightarrow \left((\widehat{x} \wedge (\widetilde{y} \Leftrightarrow \widetilde{x})) \vee (\neg\widehat{x} \wedge (\widetilde{y} \Leftrightarrow \widetilde{z}))\right)\right) & (b) \end{cases}$$

The clock of variable y is the union of the clocks of x and z, and values are determined according to the presence of x.

- *Composition:* `P` $\equiv$ `P`$_1$`|P`$_2$. The abstraction Φ of the composition operator is defined as follows:

$$\Phi \equiv \Phi_{P1} \wedge \Phi_{P2}$$

- *Restriction (or Hiding):* `P` $\equiv$ `P`$_1$ `where x`. The abstraction Φ of the restriction operator is defined as follows:

$$\Phi \equiv \exists\widetilde{x}, \exists\widehat{x} \ . \ \Phi_{P_1} \tag{1}$$

This formula may be understood as follows. The states of P are identical to the states of P_1, except that we have decided to ignore the values of $\widetilde{x}$ and $\widehat{x}$. Hence, we would like to remove from Φ_{P_1} all subformulas containing $\widetilde{x}$ or $\widehat{x}$. However, Φ_{P_1} may imply other formulas which do not use $\widetilde{x}$ and $\widehat{x}$, and are also satisfied by all states of P. This extended formula is precisely $\exists\widetilde{x}, \exists\widehat{x} \ . \ \Phi_{P_1}$ and may be found by a process of *quantifier elimination*. Conversely, it is obvious that a model of Φ can be extended to a model of Φ_{P_1}.

By applying the above rules, the following abstractions are obtained for derived constructs for clock manipulation:

- Φ(`y:= x1 ^+ x2`) $= \left(\widehat{y} \Leftrightarrow \widehat{x_1} \vee \widehat{x_2}\right) \wedge (\widehat{y} \Rightarrow \widetilde{y})$. Here, we apply the `default` abstraction rule with $\widetilde{x_1} = \widetilde{x_2} = true$ (as x_i are events), and simplify the result.
- Φ(`y:= x1 ^* x2`) $= \left(\widehat{y} \Leftrightarrow (\widehat{x_1} \wedge \widehat{x_2})\right) \wedge (\widehat{y} \Rightarrow \widetilde{y})$
- Φ(`y:= x1 ^- x2`) $= \left(\widehat{y} \Leftrightarrow (\widehat{x_1} \wedge \neg\widehat{x_2})\right) \wedge (\widehat{y} \Rightarrow \widetilde{y})$
- Φ(`x1 ^= x2`) $= \widehat{x_1} \Leftrightarrow \widehat{x_2}$

For the purpose of modularity, we also define the abstraction of processes containing subprocesses, such as in the statement $\texttt{P}_1$ `where` $\texttt{P}_2$, where $\texttt{P}_2$ is a subprocess of $\texttt{P}_1$. Let assume the following:

- $(i_1, ..., i_n)$ is the list of input parameters of $\texttt{P}_2$,
- o is a single[2] output parameter of $\texttt{P}_2$,

which represents the signature of $\texttt{P}_2$. It follows that the abstraction Φ_{P_2} is a formula composed of variables $\widehat{i_1}, \widetilde{i_1}, ..., \widehat{i_n}, \widetilde{i_n}, \widehat{o}, \widetilde{o}$. To define the abstraction of $\texttt{P}_1$ `where` $\texttt{P}_2$, we first define the abstraction of process call: `y :=` $\texttt{P}_2(x_1, ..., x_n)$ in another process, here $\texttt{P}_1$. The abstraction $\Phi($`y :=` $\texttt{P}_2(x_1, ..., x_n))$ is defined as follows:

$$(\widehat{y} = \widehat{r}) \wedge (\widetilde{y} = \widetilde{r}) \wedge \Big(\bigwedge_{i \in 1..n} (\widehat{x_i} = \widehat{z_i}) \Big) \wedge \Big(\bigwedge_{i \in 1..n} (\widetilde{x_i} = \widetilde{z_i}) \Big)$$

where $\widehat{r}, \widetilde{r}, \widehat{z_1}, \widetilde{z_1}, ..., \widehat{z_n}, \widetilde{z_n}$ are fresh variables. This abstraction only relies on the previous signature of $\texttt{P}_2$. Now, by using the previous abstraction, we finally define $\Phi(\texttt{P}_1$ `where` $\texttt{P}_2)$ as follows:

$$\begin{array}{l} \exists(\widehat{r}, \widetilde{r}, \widehat{z_1}, \widetilde{z_1}, ..., \widehat{z_n}, \widetilde{z_n}). \\ \Phi_{P1} \wedge \Phi_{P2} \qquad\qquad \bigwedge \\ (\widehat{r} = \widehat{o} \wedge \widetilde{r} = \widetilde{o}) \qquad\qquad \bigwedge \\ \Big((\widehat{z_1} = \widehat{i_1} \wedge \widetilde{z_1} = \widetilde{i_1}) ... \wedge (\widehat{z_n} = \widehat{i_n} \wedge \widetilde{z_n} = \widetilde{i_n})\Big), \end{array} \tag{2}$$

which establishes the adequate relation between the formal parameters of $\texttt{P}_2$ and the actual parameters defined in the function call within $\texttt{P}_1$.

5.4 Application to the bathtub example

By applying our abstraction to `Bathtub` (see Fig. 2), which is divided into P_1 (lines `04` to `09`) and P_2 (lines `10` to `14`) according to the process hierarchy, we obtain $\Phi_{\texttt{Bathtub}} = \Phi_{P_1} \wedge \Phi_{P_2}$, where Φ_{P_1} equals to:

$$\begin{array}{l} (\widehat{level} \Leftrightarrow \widehat{zlevel} \Leftrightarrow \widehat{faucet} \Leftrightarrow \widehat{pump} \Leftrightarrow \widehat{bzfaucet}) \\ \wedge\ (\widetilde{level} = \widetilde{zlevel} + \widetilde{faucet} - \widetilde{pump}) \\ \wedge\ \big(\widehat{zfaucet} \Leftrightarrow (\widehat{zlevel} \wedge \widetilde{zlevel} \leq 4)\big) \\ \wedge\ \big(\widehat{zfaucet} \Rightarrow \widetilde{faucet} = (\widetilde{zfaucet} + 1)\big) \\ \wedge\ (\widehat{pump} \Leftrightarrow \widehat{zpump}) \\ \wedge\ \big(\widehat{zpump} \Leftrightarrow (\widehat{zlevel} \wedge \widetilde{zlevel} \geq 7)\big) \\ \wedge\ \big(\widehat{zpump} \Rightarrow \widetilde{pump} = (\widetilde{zpump} + 1)\big) \end{array}$$

For Φ_{P_2}, we first rewrite equation at line `13/14` as follows:

```
(| y1 := true when scarce
 | y2 := y1 when overflow
 | ghost_alarm := y2 default false |)
```

Then, we obtain that Φ_{P_2} equals to:

$$\begin{array}{l} (\widehat{overflow} \Leftrightarrow \widehat{level} \Leftrightarrow \widehat{scarce}) \\ \wedge\ \big(\widetilde{overflow} \Leftrightarrow (\widetilde{level} \geq 9)\big) \\ \wedge\ (\widehat{scarce} \Leftrightarrow (\widetilde{level} \leq 0) \\ \wedge\ (\widehat{alarm} \Leftrightarrow \widehat{scarce} \Leftrightarrow \widehat{overflow}) \\ \wedge\ \widehat{alarm} \Rightarrow \big(\widetilde{alarm} \Leftrightarrow (\widetilde{scarce} \vee \widetilde{overflow})\big) \\ \wedge\ \big(\widehat{y_2} \Leftrightarrow (\widetilde{scarce} \wedge \widetilde{overflow} \wedge \widehat{scarce} \wedge \widehat{overflow})\big) \\ \wedge\ (\widehat{y_2} \Rightarrow \widetilde{y_2}) \wedge (\widehat{ghost} \Leftrightarrow (\widehat{y_2} \vee \widehat{false})) \\ \wedge\ \Big(\widehat{ghost} \Rightarrow \big((\widehat{y_2} \wedge (\widehat{ghost} \Leftrightarrow \widetilde{y_2})) \vee (\neg\widehat{y_2} \wedge \neg\widetilde{ghost})\big)\Big) \end{array}$$

[2]Here, we consider a single output only for the sake of simplicity. The same reasoning strictly applies for several outputs.

5.5 Concretisation

Let us recall that $X = \{x_1, \dots x_n\}$ denotes the set of all P variables. Intuitively, a valuation satisfying Φ captures the numerical and Boolean values of signals at a given logical instant. Given a valuation $v = (\widehat{\ }, \widetilde{\ })$, where all variables have been assigned some values, we first construct a set of events whose values are assigned accordingly: $S_{valid}(v) = \{S \in \mathcal{E}_X | \forall i, S(i) = if\ (\widehat{x_i} = false)\ then\ \bot\ else\ \widetilde{x_i}\}$. The set of all "valid" events is defined as $S_{valid}(\Phi) = \cup_{v \models \Phi} S_{valid}(v)$. Finally, the concretisation of Φ is the set of traces whose instantaneous values always verify Φ :

$$\Gamma(\Phi) \quad = \quad \{T \in \mathcal{T}_X | \forall t, T(t) \in S_{valid}(\Phi)\} \tag{3}$$

Our abstraction is sound, in the sense that it preserves the behaviors of the abstracted programs: if a property is *true* on the abstraction, then it is also the case on the program. A proof of its soundness is given in [11].

5.6 Properties

Let P be a SIGNAL process and Φ its abstraction. Assume that we can prove formulas of the form $\Phi \Rightarrow \Pi$, where Π is a formula on the atoms of Φ. It is clear that Φ and $\Phi \wedge \Pi$ have the same models. Some such formulas have the property that they are abstraction of SIGNAL processes. These processes can be composed with P to the benefit of the SIGNAL compiler without modifying the semantics of P.

The properties we are interested in are clock emptiness: $\widehat{x}$ = false, which gives the equivalent of dead code elimination, and clock inclusion: $\widehat{x} \Rightarrow \widehat{y}$ or clock equivalence: $\widehat{x} \Leftrightarrow \widehat{y}$, which allow simplification of the control code. There are two strategies for finding such properties. The first one consists in guessing Π and proving $\Phi \Rightarrow \Pi$ with the help of an SMT solver, by showing that $\neg(\Phi \Rightarrow \Pi)$ is unsatisfiable. The second strategy consists in asking the SMT solver to construct the set of (Boolean) models of Π, which is finite, and to scan it to identify interesting properties. For instance, the algorithm for finding empty clocks is to start from the set of all clocks, to examine each model in turn, removing a clock as soon as it appears to be *true* in the current model. This is the approach we have adopted in our implementation.

6 Implementation

We present an implementation of the previous abstraction and the way relevant properties are inferred. Our solution promotes a modular construction of this abstraction and its analysis.

6.1 Tools

The implemented tools follow Fig. 1. The box referred to as "Abstraction of P" in this figure is achieved with the SYNC2SMT tool. Its output is given to an *ad hoc* SMT solver, which integrates the concretization of inferred properties.

SYNC2SMT (5kLOC in Ocaml) basically implements the translation developed in Section 5 : after a parsing phase, the internal representation of a SIGNAL program is translated into a bunch of `smtlib`[3] files, including a special "driver" file. Such a file is used as an input to our *ad hoc* SMT solver. Note that our parser currently recognizes only a subpart of the grammar described in `http://www.irisa.fr/espresso/Polychrony/Signal-bnf.php`.

There are two reasons for not using an off-the-shelf SMT solver like Yices or Z3. The first one is that we need more than a `sat` or `unsat` answer. Our solver must construct the set of all models of a satisfiable formula and return it for inspection. Usually, an SMT solver constructs just one model (this is enough for proving satisfiability), which can be retrieved or not depending on the solver. It is clear that our solver is less efficient than highly optimized softwares like Yices or Z3. However, since we trade just one call to a slow solver against many calls to a fast solver, the overall comparison is not obvious. Another point is that since the solver code is available to us, we have been able to implement the property search inside it, thus avoiding costly pretty printing and parsing.

[3] `http://www.smtlib.org/`

Our SMT solver proceeds by constructing a semantic tableau [26], i.e., a tree whose nodes are decorated by subformulas of the root formula. A branch of the tree is closed if it contains a formula and its negation, or if the conjunction of its atomic formulas is unsatisfiable in the underlying theory, in our case, linear or integer programming. The tree construction rules are such that from each open branch, one can extract a model of the root formula. From then on, it is a simple matter to scan the open branches and extract clock properties.

6.2 Modularity

While current SMT solvers are highly optimized tools, they may still take exponential time on large problems. It is therefore necessary to take advantage of the modular features of SIGNAL to improve the analysis efficiency. The key to this approach is formula (1), which allows the elimination of local variables when analyzing subprocesses.

Going from Φ_{P_1} to Φ in (1) is a process of quantifier elimination, which is trivial for booleans:

$$\exists b.\Phi(b) \equiv \Phi(true) \vee \Phi(false).$$

However, Φ usually contains many subformulas of the form $\widehat{x} \Leftrightarrow bexp$ (see Section 5.4 for examples). Elimination of $\widehat{x}$ consists simply in replacing it everywhere by $bexp$, a process akin to Gaussian elimination.

There are many quantifier elimination algorithms for reals, the simplest (but the less efficient) being Fourier-Motzkin elimination [27]. Quantifier elimination for integers is much more difficult, and may need the introduction of other operators like integer division or modulo. To apply this method, our SMT solver has been extended with a quantifier elimination command, and several commands to manipulate a stack of formulas.

Let us consider the simple case of a program of the form `P1 where P2`. From (2), the output of SYNC2SMT consists first of the abstraction of P_2. A "driver" file first acquires the P_2 file and executes elimination of the local variables. Another file contains the abstraction of P_1, augmented with a system of equations that identifies the actual arguments of P_2 in P_1 to the formal arguments of P_2. The tool constructs the conjunction of the two formulas, checks satisfiability, and deduces clock properties from the resulting models.

In more complex examples, one can apply the same algorithm bottom-up to a tree of processes. The properties found in this way for the top process can be plugged top-down into the subordinate processes. One may have to use renaming to avoid symbol collision or capture.

7 Application to illustrative examples

We discuss the application of the previous abstraction on sample SIGNAL programs, considered as basic patterns, for improving their static analysis (Section 7.1) and the subsequent automatic code generation (Section 7.2). Then, we give a detailed illustration on the `Bathtub` example (Section 7.3).

7.1 Some relevant program patterns

We present a few SIGNAL program patterns for which our abstraction helps in detecting some clocks anomalies. Such properties cannot be detected currently by the SIGNAL compiler because they involve numerical expressions, which are not addressed by a Boolean abstraction. Our abstraction allows their easy detection.

For sake of simplicity, the illustrated programs are made small. But, the reader should have in mind that such clock properties can potentially occur in more complex programs.

7.1.1 Program patterns involving exclusive clocks

The sample processes mentioned in this section involve signals with exclusive clocks, i.e., signals that never occur at the same time.

1. In the following process `Addition`, the signals `aa` and `bb`, respectively defined at lines `05` and `06`, never occur at the same time, while the converse is necessary (according to the semantics of instantaneous functions in SIGNAL) for a correct addition at line `04`.

```
-----------------------------------------------
01: process Addition =
02: ( ? integer a, b, treshold;
03:   ! integer c; )
04: (| c := aa + bb
05:  | aa := a when (treshold > 7)
06:  | bb := b when (treshold < 4 )
07:  |)
08: where
09:   integer aa, bb;
10: end;
-----------------------------------------------
```

2. For a similar reason, in the following process `AdditionBis`, the addition of signals `b` and `c`, respectively defined at lines `04` and `05`, cannot be achieved in a correct way. Indeed, the conditions specified for the definitions of `b` and `c` are exclusive. Note that the difference between `Addition` and `AdditionBis` is mainly syntactical.

```
-----------------------------------------------
01: process AdditionBis =
02:   ( ? integer a;
03:     ! integer d; )
04:   (| b := a when (a > 1)
05:    | c := a when not (a > 0)
06:    | d := b + c
07:    |)
08: where
09:   integer b, c;
10: end;
-----------------------------------------------
```

3. The last sample process shown below, involves signals with exclusive clocks, `bmin` and `bmax`, defined respectively at lines `04` and `05`. But, another signal `binterval`, defined at line `06` as an under-sampling over `bmin` and `bmax`, has an empty clock because the two signals never occur at the same time.

```
-----------------------------------------------
01: process Interval =
02:  ( ? integer a;
03:    ! event binterval; )
04:  (| bmin := true when (a < 3)
05:   | bmax := true when (a > 11)
06:   | binterval := bmin when bmax
07:   |)
08: where
09:  event bmin, bmax;
10: end;
-----------------------------------------------
```

7.1.2 Program patterns involving identical clocks

Here, we show two sample processes involving signals with identical clocks. This is fixed by our abstraction while the Boolean abstraction of the SIGNAL compiler does not enable it.

1. In the following process, named `AdditionTer`, the addition of signals `b` and `c`, respectively defined at lines `04` and `05`, is actually correct. Indeed, the conditions specified for the definitions of these two signals are proved to be equivalent.

```
-----------------------------------------------
01: process AdditionTer =
02:  ( ? integer a;
```

```
03:    ! integer d; )
04:  (| b := 5+a when (a > 0)
05:   | c := 6+a when (a >= 1)
06:   | d := b + c
07:   |);
08: where
09:  integer b, c;
10: end;
-----------------------------------------------
```

2. The process `Game` shown below exhibits similar clock properties. More precisely, the product at line `09` of the input signal `amount` and the local signal `factor` defined at lines `07--08`, requires that both signals have the same clock.

   This is established by a careful interpretation of the `modulo` operator (used at line `06`). Indeed, the expression *nvisit* `modulo` 2 is abstracted by $\exists q, r \in \mathbb{N}, \text{s.t. } r = nvisit - 2q \ \wedge \ 0 \leq r \leq 1 \ \wedge \ 2q \leq nvisit \leq 2q + 1$, where q and r respectively denote the quotient and rest of integer division.

```
-----------------------------------------------
01: process Game =
02:  ( ? integer amount;
03:    ! integer profit; )
04:  (| nvisit := ((nvisit$1 init 0) + 1)
05:                when (^amount)
06:   | st := nvisit modulo 2
07:   | factor := (15 when (st=0)) default
08:               (0 when (st=1))
09:   | profit := factor*amount
10:   |)
11: where
12:  integer st, factor, nvisit;
13: end;
-----------------------------------------------
```

7.2 Impact on code generation

Our abstraction is also usable for optimizing the control structure of the code generated by the SIGNAL compiler. As discussed in Section 4, the clock hierarchy resulting from the static analysis of programs has a strong impact on the quality of the generated code. Since clocks are considered as trigger events for the actions described in a program, they are translated as conditional statements in generated code, e.g., in C.

Given two clocks `clk_1` and `clk_2` such that `clk_2` is a sub-clock of `clk_1`, the corresponding code is sketched in Fig. 5: the conditional statement corresponding to `clk_2` is embedded in that associated with `clk_1` to reflect the clock inclusion. By this way, whenever the triggering condition of `clk_1` is *false*, there is no need to test the triggering condition of `clk_2` because it is necessarily *false* due to the clock inclusion. Avoiding such tests optimizes the execution of generated code. Note that a major advantage of the multi-clock model addressed by SIGNAL is to avoid the systematic trigger testing inherent to synchronized embedded systems with a global clock. This reduces the computation overhead resulting from the repeated wake up of computation nodes on the global clock tick in order to check whether or not they are active.

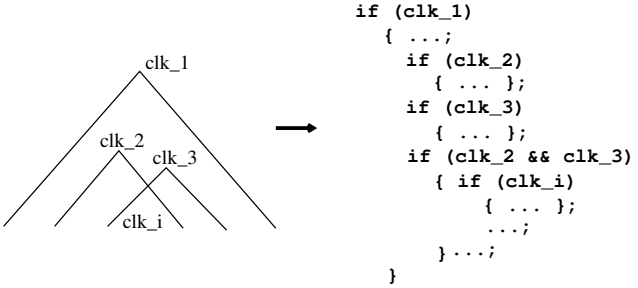

Figure 5: Clock hierarchy-based code generation.

Currently, when clocks are defined by numerical expressions, the static analysis of the SIGNAL compiler fails to optimize the control structure in the way discussed above.

Let us consider the sample process, named `Inclusion`, as follows.

```
--------------------------------------------------
01: process Inclusion =
02:  ( ? integer a;
03:    ! integer d, e; )
04:  (| b := 5+a when ((a > 3) and (a < 7))
05:   | c := 6+a when ((a > 1) and (a < 11))
06:   | d := 42 when (b ^* c)
07:   | e := 52 when (b ^+ c)
08:   |)
09: where
10:  integer b, c;
11: end;
--------------------------------------------------
```

The clock of signal `b` is a subset of that of `c`. But currently, the clock hierarchy computed by the SIGNAL compiler is depicted in Fig. 6. While the clocks of `b` and `c` appear to be sub-clocks of the clock of `a`, the clock hierarchy between `b` and `c` is not reflected. This leads to a control structure in generated code where the trigger testing related to `b` is always performed, even though that of `c` is *false* while it is unnecessary.

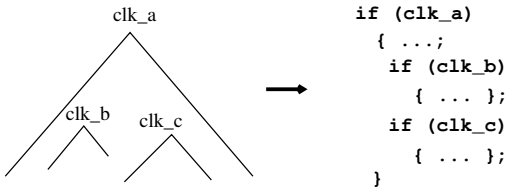

Figure 6: Clock hierarchy for `Inclusion` process.

Our abstraction is able to prove the clock inclusion between `b` and `c`, with the following reasoning. A clock $\widehat{x}$ is included in another clock $\widehat{y}$ if the property $\widehat{x} \Rightarrow \widehat{y}$ is *true* in all models. Clock $\widehat{x}$ is equivalent to clock $\widehat{y}$ if both $\widehat{x} \Rightarrow \widehat{y}$ and $\widehat{y} \Rightarrow \widehat{x}$ are *true*.

When all inclusions have been identified, one can construct a graph whose vertices are the clocks and whose edges represent the inclusion relations. The strongly connected components (SCC) of this graph represent classes of equivalent clocks, and the reduced graph, which is acyclic, represents the clock inclusion hierarchy. As a particular case, if this graph has a maximum (an SCC without successors) this SCC contains the largest clock of the whole process. The set of SCCs and the reduced graph can easily be constructed by an algorithm due to Tarjan [28], which has been implemented in our tool (more precisely in the solver part). As a matter of fact, since inclusion is transitive, the SCCs of the clock graph are cliques. However, we do not believe that this property can be used to improve on the complexity of Tarjan's algorithm. Note also that as soon as the maximal SCC has more than one element, the largest clock cannot be identified by searching for clocks without successors. Hence, the construction of SCCs is necessary. As a final remark, if the SCC graph has more than one extrema, the program has no sequential implementation.

In the `Inclusion` process above, one finds three SCCs, $\{\widehat{b}, \widehat{d}\}$, $\{\widehat{c}, \widehat{e}\}$ and $\{\widehat{a}\}$, and each SCC is included in the next one. It follows that $\widehat{a}$ is the process largest clock, which provides the clock inclusion hierarchy depicted in Fig. 7.

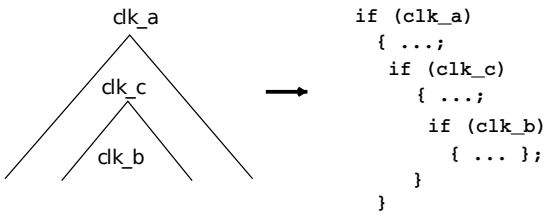

Figure 7: Optimized clock hierarchy for `Inclusion`.

The ability to compute the above clock inclusions is a very useful information, which can be exploited to efficiently construct clock hierarchy for SIGNAL programs based on arborescent canonical forms of clocks [6]. The identification of a master clock in a program relies on that clock hierarchy.

7.3 Application to the bathtub example

We consider the `Bathtub` program given in Fig. 2 to illustrate how relevant properties are identified and checked against its abstraction. By making these properties explicit in the program, we show a noticeable amelioration of both its static analysis and code generation by the SIGNAL compiler.

Given the formula $\Phi_{\texttt{Bathtub}}$ obtained previously in Section 5.3, as the abstraction of the bathtub SIGNAL specification, the main properties of interest are the following:

1. `pump` and `faucet` have disjoint clocks: $\neg(\widehat{faucet} \wedge \widehat{pump})$,
2. The water cannot overflow and be scarce at the same time: $\neg\big(\widehat{\widetilde{scarce}} \wedge \widehat{\widetilde{overflow}} \wedge \widehat{scarce} \wedge \widehat{overflow}\big)$,
3. `alarm` and `level` have the same clock: $\widehat{alarm} \Leftrightarrow \widehat{level}$.

Some of these properties are currently inferred directly from $\Phi_{\texttt{Bathtub}}$ by our considered SMT solver. It is the case of properties 1) and 3). However, note that property 2) could also be inferred provided an extension of the current implementation of the solver so that various combinations of Boolean variables can be checked. Here, for more convenience, we reason on isolated parts of $\Phi_{\texttt{Bathtub}}$, which are relevant to a given property. But, since automating such an operation on an abstraction is generally not easy, our implementation currently reasons on the whole abstraction.

These properties are easily verified on the abstraction of `Bathtub` process. As a result, their corresponding concretisations can be safely composed with `Bathtub` without changing its semantics. Possible concretisations of the above properties in SIGNAL are as follows:

1. `faucet ^* pump ^= ^0`
2. `true when scarce when overflow ^= ^0`
3. `alarm ^= level`

By composing these statements with `Bathtub`, one obtains the semantically equivalent process, named `Bathtub_Bis`, shown in the following:

```
-------------------------------------------------
01:process Bathtub_Bis =
02:(?
03: ! integer level; boolean alarm, ghost_alarm; )
04:(|(| level := zlevel + faucet - pump
    ...
13:   | ghost_alarm:=(true when scarce when overflow)
13b:                  default false |)
14: |(| true when scarce when overflow ^= ^0
15:   | faucet ^* pump ^= ^0
16:   | alarm ^= level |) |)
17: where
18:  integer zlevel,zfaucet,zpump,faucet,pump;
19   boolean overflow,scarce;
20:end;
-------------------------------------------------
```

The result of its analysis performed by the compiler is now as follows:

```
-------------------------------------------------
01: (| CLK_ghost_alarm := ^ghost_alarm
02:  | CLK_ghost_alarm ^= ghost_alarm
03:  | (| ghost_alarm := not CLK_ghost_alarm |)
04:  |);%^0 ^= level ^= alarm
04b         ^= zlevel ^= zfaucet ^= zpump
05:       ***WARNING: null clock signals%
-------------------------------------------------
```

The whole set of constraints inferred by the compiler is now restricted to the fact that the `ghost_alarm` signal is always equal to `false`. The compiler has also detected that the clocks of

the other signals are all empty (lines 04/04b). Finally, the corresponding generated code is provided below, where the dead code is avoided.

```
-----------------------------------------------------
01:   { ghost_alarm = FALSE;
02:    /* produce output value
03:         for the signal ghost_alarm */ } ...
-----------------------------------------------------
```

Sections 7.1, 7.2 and 7.3 demonstrate the relevance of our abstraction for analyzing clock properties that combine both logical and numerical expressions. For instance, checking the mutual exclusion between multiple computation nodes whose activation conditions consist of such clocks, is useful to address sharing problems in a GALS system. In addition, establishing that some nodes or events in a system never occur, via empty clocks, can serve to guarantee that undesired behaviors never happen, or conversely to detect that some expected behaviors are never observed. Concerning the code generated automatically by the SIGNAL compiler, the gain expected in terms of optimizations is also important. On the one hand, dead code elimination is made possible thanks to information resulting from the analysis of our abstraction. It is usually of high importance in compilers [29]. On the other hand, the control conditions of the code are better organized thanks to their evaluation in the abstraction. As a result, optimized control structures can be derived, as it is done in [30] by identifying *regions* in a control flow graph.

7.4 On the scalability of our approach

Beyond all examples mentioned in this paper, we have experimented further ones, including the *dining philosophers* program provided in [24], which is relevant enough to assess the scalability of our tool-chain, but which strains the present capabilities of our SMT solver.

Among applicable solutions that already hold for our approach in case of large programs to be addressed, we suggest the systematic use of modularity to divide-and-conquer such programs. As a matter of fact, given a property to be checked (or to be inferred) in an SMT formula F resulting from the translation of a program, one can restrict the analysis to the sub-formulas F_i of this formula, which are only required for the reasoning. Whenever a property is valid for F_i, it will be also valid for F. Currently, identifying such sub-formulas is done only manually.

For the aforementioned *dining philosophers* program, which is around one hundred and seventy lines of code in SIGNAL, our translation tool automatically generates (in less than a second) an abstraction in the "smt2" format composed of: four hundred and fourty variables and, four hundred and eighty six clauses. Since this generated abstraction is not currently tractable by our SMT solver, we manually applied a divide-and-conquer strategy to check that two adjacent philosophers cannot simultaneously eat because only one of them can hold their shared fork at any time.

8 Conclusion

In this paper, we presented an enhancement of the compilation of synchronous dataflow programs with a combined numerical-Boolean abstraction. We considered SIGNAL language as an illustrative language. The analysis and code generation achieved by its compiler, which is based on a Boolean abstraction, has been extended in a modular way by defining a sound and more expressive abstraction. This makes it possible to suitably address both numerical and logical properties specified via abstract clock relations and data dependencies.

Clocks play a central role in SIGNAL: they fundamentally express the control in programs and typical properties of embedded systems, such as reactivity or determinism, are dealt with by analyzing clock relations. Moreover, their related properties are extensively exploited by the SIGNAL compiler for optimizing the automatic code generation process. We showed via our approach, in a pragmatic way, how the new abstraction combined with SMT solving infers very useful information, which strongly help the compiler to solve more clock constraints and generate high-quality code, *e.g.*, by avoiding dead code. Several sample examples have been presented in order to exhibit the add-on of our solution.

To implement the whole approach, we developed a translator of synchronous programs towards the standard input format of SMT solvers, and an *ad hoc* SMT solver that integrates advanced functionalities to cope with the issues of interest in this work. These tools are just proof-of concept implementations; we do not claim that they can be used on lifesize programs in their present state. Improvements are needed in four directions:

- replace our home-made SMT solver by a state-of-the-art one, provided that its source code is available and that it can be be adjusted to implement the supplementary facilities we need;
- improve the Sync2smt translator to obtain a more compact abstraction;
- implement an interval pre-analysis to get value ranges for numerical variables and thus provide a better abstraction for delays;
- systematically use modularity to divide-and-conquer large programs.

References

[1] A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone, "The synchronous languages twelve years later." in *Special issue on Embedded Systems, IEEE*, 2003.

[2] N. Halbwachs, "A synchronous language at work: the story of Lustre," in *3th ACM-IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE'05)*, Verona, Italy, july 2005.

[3] G. Berry, "The foundations of Esterel," in *Proof, Language and Interaction: Essays in Honour of Robin Milner*. MIT Press, 2000.

[4] P. Le Guernic, J.-P. Talpin, and J.-C. Le Lann, "Polychrony for System Design," *Journal for Circuits, Systems and Computers*, vol. 12, no. 3, pp. 261–304, April 2003.

[5] R. Bryant, "Graph-based algorithms for boolean function manipulation." *IEEE transactions on computers*, vol. C-35, no. 8, pp. 677–691, August 1986.

[6] T. Amagbegnon, L. Besnard, and P. Le Guernic, "Arborescent canonical form of Boolean expressions," INRIA, Tech. Rep. 2290, June 1994. [Online]. Available: http://www.inria.fr/rrrt/rr-2290.html

[7] N. Halbwachs, F. Lagnier, and C. Ratel, "Programming and verifying real-time systems by means of the synchronous data-flow programming language Lustre." *IEEE Transactions on Software Engineering, Special Issue on the Specification and Analysis of Real-Time Systems*, September 1992.

[8] B. Jeannet, "Dynamic partitioning in linear relation analysis. application to the verification of reactive systems," *Formal Methods in System Design*, vol. 23, no. 1, pp. 5–37, July 2003.

[9] P. Schrammel, "Logico-Numerical Verification Methods for Discrete and Hybrid Systems," Ph.D. dissertation, Université de Grenoble, 2012.

[10] B. A. Jose and S. K. Shukla, "An alternative polychronous model and synthesis methodology for model-driven embedded software," in *Proceedings of the 2010 Asia and South Pacific Design Automation Conference*, ser. ASPDAC '10. Piscataway, NJ, USA: IEEE Press, 2010, pp. 13–18. [Online]. Available: http://dl.acm.org/citation.cfm?id=1899721.1899725

[11] A. Gamatié and L. Gonnord, "Static analysis of synchronous programs in signal for efficient design of multi-clocked embedded systems," in *International conference on Languages, Compilers and Tools for Embedded Systems, LCTES'11*, Chicago, USA, Mar. 2011.

[12] A. Biere, M. Heule, H. van Maaren, and T. Walsh, *Handbook of Satisfiability: Volume 185 Frontiers in Artificial Intelligence and Applications*. Amsterdam, The Netherlands, The Netherlands: IOS Press, 2009.

[13] P. Schrammel and B. Jeannet, "From hybrid data-flow languages to hybrid automata: A complete translation," in *Hybrid Systems: Computation and Control*. ACM, 2012, pp. 167–176.

[14] L. Gonnord and N. Halbwachs, "Combining widening and acceleration in linear relation analysis," in *13th International Static Analysis Symposium, SAS'06*, Seoul, Korea, Aug. 2006.

[15] F. Besson, T. Jensen, and J.-P. Talpin, "Polyhedral analysis for synchronous languages," in *Proceedings of the 6th International Symposium on Static Analysis, volume 1694 of Lecture Notes in Computer Science*. Springer-Verlag, September 1999, pp. 51–68.

[16] M. Nanjundappa, M. Kracht, J. Ouy, and S. Shukla, "Synthesizing embedded software with safety wrappers through polyhedral analysis in a polychronous framework," in *Electronic System Level Synthesis Conference (ESLsyn), 2012*, june 2012, pp. 24 –29.

[17] M. Nebut, "Specification and analysis of synchronous reactions," *Formal Aspects of Computing*, vol. 16, no. 3, pp. 263–291, august 2004.

[18] G. Hagen and C. Tinelli, "Scaling up the formal verification of lustre programs with smt-based techniques," in *FMCAD '08: Proceedings of the 2008 International Conference on Formal Methods in Computer-Aided Design*. Piscataway, NJ, USA: IEEE Press, 2008, pp. 1–9.

[19] A. Gamatié, T. Gautier, and P. Le Guernic, "Towards static analysis of SIGNAL programs using interval techniques." in *Synchronous Languages, Applications, and Programming (SLAP'06)*, March 2006.

[20] A. Gamatié, T. Gautier, and L. Besnard, "An Interval-Based Solution for Static Analysis in the SIGNAL Language," in *15th Annual IEEE International Conference and Workshop on Engineering of Computer Based Systems (ECBS'2008), Belfast, Northern Ireland*, April 2008, pp. 182–190.

[21] Y. Bai, J. Brandt, and K. Schneider, "Smt-based optimization for synchronous programs," in *Proceedings of the 14th International Workshop on Software and Compilers for Embedded Systems*, ser. SCOPES '11. New York, NY, USA: ACM, 2011, pp. 11–20. [Online]. Available: http://doi.acm.org/10.1145/1988932.1988935

[22] B. A. Jose, A. Gamatié, J. Ouy, and S. K. Shukla, "SMT Based False Causal loop Detection during Code Synthesis from Polychronous Specifications," in *ACM/IEEE Ninth International Conference on Formal Methods and Models for Codesign (MEMOCODE)*, 2011, pp. 109 –118.

[23] B. A. Jose, A. Gamatié, M. Kracht, and S. K. Shukla, "Improved False Causal Loop Detection in Polychronous Specification of Embedded Software, Research report," 2011. [Online]. Available: http://hal.inria.fr/inria-00637582

[24] A. Gamatié, *Designing Embedded Systems with the* SIGNAL *Programming Language: Synchronous, Reactive Specification*. Springer, New York, 2009.

[25] P. Le Guernic and T. Gautier, *Advanced Topics in Data-Flow Computing*. Prentice-Hall, J.-L. Gaudiot and L. Bic eds., 1991, ch. Data-Flow to von Neumann: the SIGNAL approach, pp. 413–438.

[26] R. M. Smullyan, *First Order Logic*. Dover, 1968.

[27] A. Schrijver, *Theory of linear and integer programming*. NewYork: Wiley, 1986.

[28] R. E. Tarjan, "Depth first search and linear graph algorithms," *SIAM J. on Computing*, vol. 1, pp. 146–160, 1972.

[29] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM Trans. Program. Lang. Syst.*, vol. 13, pp. 451–490, October 1991. [Online]. Available: http://doi.acm.org/10.1145/115372.115320

[30] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Trans. Program. Lang. Syst.*, vol. 9, pp. 319–349, July 1987. [Online]. Available: http://doi.acm.org/10.1145/24039.24041