Static Analysis of Synchronous Programs in Signal for Efficient Design of Multi-Clocked Embedded Systems

Abdoulaye Gamatié
LIFL - UMR CNRS/Lille1 8022 and Inria,
40 avenue Halley, Park Plaza - Bâtiment A,
59650 Villeneuve d'Ascq, France
abdoulaye.gamatie@lifl.fr

Laure Gonnord
LIFL - UMR CNRS/Lille1 8022,
Cité scientifique - Bâtiment M3
59655 Villeneuve d'Ascq Cedex, France
laure.gonnord@lifl.fr

Abstract

In this paper, we propose a sound abstraction for an efficient static analysis of synchronous programs describing multi-clock embedded systems in SIGNAL. This abstraction combines the Boolean theory and numeric interval approximation to adequately address clock relations defined as combinations of logical and numerical expressions. Through a few examples, we show how the proposed 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 the SIGNAL compiler, *e.g.*, a code with smaller footprint, or a code executed more efficiently thanks to optimizations enabled by the new abstraction.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification—Formal methods, Correctness proofs; C.3 [*Computer Systems Organization*]: Special-Purpose and Application-Based Systems—Real-time and embedded systems

General Terms Verification, Design, Reliability

Keywords Embedded systems, static analysis, abstraction, abstract clocks, synchronous programming, compilation

1. Introduction

Modern embedded systems increasingly include multiple clock domains in both their hardware and software parts. Typical examples are multi-processor system-on-chip (MPSoCs) used in consumer electronics to achieve high performance and energy efficiency by implementing dynamic voltage and frequency scaling (DVFS) [27]. The frequencies of different computing elements, such as processors, dynamically vary to ensure as high as possible the quality of service (QoS) of a system. The multiple clock domains resulting from this frequency variation offer a flexible way to address a global performance/energy tradeoff in an MPSoC, via local decisions about increasing or decreasing the frequency of a processor. A similar observation is made at system level when designing embedded applications by several autonomous functional blocks or modules, running concurrently on different computation nodes, for instance multi-rate tasks. Indeed, the computation overhead is reduced when modules are activated *only when strictly required*: a module does not need to wake up frequently, according to a global clock of a system, to check whether or not it has to execute; instead, this should be dictated by a local clock of its computation node.

An interesting design solution regarding the above examples consists in using *globally asynchronous and locally synchronous* (GALS) architectures [9, 28] as illustrated in FIG. 1. Each computation node holds its own clock providing a local (synchronous) vision of time. The time scale according to which its associated events are observed is not necessarily identical to those of the other nodes. In FIG. 1, events are represented by bullets labelled with the occurrence rank according to their corresponding time scale (a horizontal line). The interactions between the three illustrated nodes can be represented using synchronization relations between event occurrences, *e.g.*: first event occurrence (tagged "0") of **node 1** and third event occurrence (tagged "2") of **node 2**, second event occurrence of **node 1** and second event occurrence of **node 3**, etc. From an overview of a system, these relations only yield a partial occurrence ordering of all observed events; while focusing on a node, all its local events are totally ordered with respect its clock.

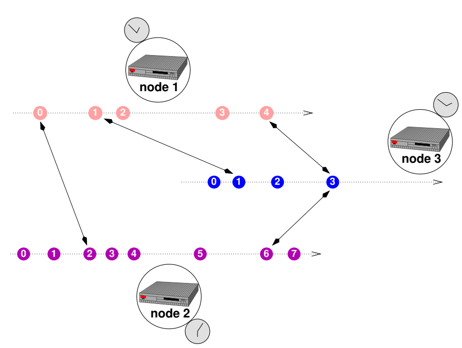


Figure 1. A multi-clocked GALS system.

1.1 Synchronous approach for multi-clocked system design

Dealing with the correct design of embedded systems with multiple clock domains at the hardware level is very complex because of various factors, *e.g.*, noise and jitter on clock signals, and skew between data signal and clock signals. As a solution, the problem can be addressed rather at a higher level. The *abstract clock* notion provided by synchronous languages [4] offers the opportunity to

LCTES'11 12-14 April, Chicago, IL, USA.

suitably address the problem. An abstract clock of a signal is a discrete set of logical instants at which events occur on the signal. Typically, the event occurrences at a node in FIG. 1 can be characterized by an abstract clock. Then, the synchronization relations between events of different nodes will be described as clock relations. So, abstract clocks play a central role in the design of multi-clocked embedded systems.

Some of these languages are LUSTRE [20], ESTEREL [5] and SIGNAL [26]. They have been proposed in the early 80's as an answer to the reliable development of safety critical embedded systems. Today, 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 can mention their mathematical foundation that favors a precise semantics of programs, the ability to trustworthily reason on program properties, the possibility to automatically generate correct-by-construction target implementations from programs, and finally a wide range of supporting tools.

1.2 Multi-clocked models in SIGNAL

The design of multi-clocked embedded systems with a synchronous language such as LUSTRE assumes a global clock providing the time scale for all computation nodes. In terms of set of instants, the activation clocks of nodes are strict subsets of the global clock. While this synchronized model of a system is suitable for guaranteeing determinism, it enforces a monolithic vision of the whole design so that one cannot focus on the activity of a given node regardless of the reference (or global) clock of the system.

The design model adopted in the SIGNAL language is different from that of LUSTRE and ESTEREL by enabling the description of computation nodes without assuming any global clock in a system. It is referred to as *polychronous* model [26]. Abstract clocks particularly play a fundamental role in the polychronous design of embedded systems in SIGNAL. They are used to describe all the control part: activation triggering of components and interaction between different components via clock relations, as illustrated in FIG. 1. The control flow resulting from these clock relations also serves to derive an optimized control structure in generated code. Thus, the quality of the clock analysis has a strong impact on design correctness and efficiency. POLYCHRONY, the design environment of SIGNAL allows a designer to specify, analyze and automatically generate code for multi-clocked systems, such as GALS. Beyond the usual syntax or type checking, its compiler implements powerful static analysis and optimizations, allowing for a correct and efficient code generation. This analysis relies on a Boolean abstraction of programs, internally represented as *binary decision diagrams* (BDD) [8] for an efficient reasoning [2].

1.3 Problem statement: analysis of numerical properties

When the static analysis performed by the SIGNAL compiler addresses the clock properties of a program, defined with numerical expressions, the current Boolean abstraction loses some 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. For instance, such a situation arrives when defining in SIGNAL the activation clocks of a system as sets of events that occur when the values of some data signals satisfy a numerical property: activation of a (rescue) computation node in a fault-tolerant GALS system when an data signal from an executing node reach some particular value, activation of a node whenever another computation node produces a periodic event, etc. In order to address suitably this kind of property, an new abstraction is required, which fully takes into account the numerical part beyond the Boolean part of SIGNAL programs.

1.4 Contribution of this paper

We propose a sound Boolean-interval abstraction for the static analysis of synchronous programs defining multi-clocked embedded systems in SIGNAL. Our solution permits an analysis that significantly enhances the quality of the subsequent code generated automatically by the compiler, *e.g.*, a code with smaller footprint, or a code executed more efficiently thanks to further optimizations. 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, abstracted as an interval. Given the level of performance reached by recent progress in Satisfiability Modulo Theory (SMT) [7], we use an SMT solver to implement the reasoning on the new abstraction. We show through a few examples, how relations between abstract clocks defined with numerical and logical expressions can be 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 hierarchical control of computation node activations via clock inclusion.

1.5 Outline

The remainder of this paper is organized as follows. Section 2 gives an overview of SIGNAL and illustrates a simple yet typical example. Section 3 discusses the current limitations of the static analysis achieved by the SIGNAL compiler, regarding clock analysis and code generation. Section 4 exposes an abstraction for improving this static analysis by using first-order logic formulas. Section 5 presents an implementation of our solution, with an illustration on a few examples. Section 6 discusses the proposed approach with respect to related work. Finally, Section 7 gives concluding remarks.

2. Overview of the SIGNAL language features

SIGNAL [26] [15] 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.

2.1 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. The formal semantics is introduced in Section 2.2.

- *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

A. Gamatié and Gonnord , accepted to ACM Languages, Compilers, Tools and Theory for embedded Systems (LCTES 2011, Chicago) ---------------------- *©2011 ACM*

Table 1. Trace semantics for SIGNAL primitives.

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

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 `z, y, x` 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` uppermost, otherwise that of `y`.
- *Composition:* $\mathtt{P} \equiv \mathtt{P}_1 | \mathtt{P}_2$ where $\mathtt{P}_1$ and $\mathtt{P}_2$ are processes. It denotes the union of equations defined in processes, leading to the conjunction of the constraints associated with these processes. The composition operator is commutative and associative.
- *Restriction (or Hiding):* $\mathtt{P} \equiv \mathtt{P}_1$ `where x`, where $\mathtt{P}_1$ and `x` are a process and a signal. It states that `x` is a local signal of process $\mathtt{P}_1$. The process `P` holds the same constraints as $\mathtt{P}_1$.

Some useful derived constructs 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`.

2.2 A denotational semantics of the language

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 $\mathcal{E}_{X_1} = X_1 \rightarrow \mathcal{D}^{\perp}_{X_1}$, 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 represented 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 $\mathcal{T}^{\perp}_{X1} : \mathbb{N} \rightarrow \mathcal{E}_{X1}$, 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, $\mathcal{T}_{\emptyset} = \mathbf{1} = \mathbb{N} \rightarrow \mathcal{E}_{\emptyset}$.

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 $X_2.T : \mathbb{N} \rightarrow \mathcal{E}_{X_2}$, 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 also define the trace restriction (or *projection*) of set of traces $\mathcal{T}$ to $X \subseteq X_{\mathcal{T}}$ 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 $[\![\mathtt{P}]\!]$. Each SIGNAL primitive construct defines an elementary process whose trace semantics is given in Table 1.

2.3 Example: a bathtub model in SIGNAL

The simple SIGNAL process shown in FIG. 2 specifies the status of a *bathtub* [6]. It has no input signal (line `02`), but has three output signals (line `03`).

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.

```
-------------------------------------------------
process Bathtub =
(?
 ! integer level; boolean alarm, ghost_alarm; )
(|(| level := zlevel + faucet - pump
   | zlevel := level$1 init 1
   | faucet := zfaucet + (1 when zlevel <= 4)
   | zfaucet := faucet$1 init 0
   | pump := zpump + (1 when zlevel >= 7)
   | zpump := pump$1 init 0 |)
 |(| overflow := level >= 9
   | scarce := 0 >= level
   | alarm := scarce or overflow
   | ghost_alarm:= (true when scarce when overflow)
               default false |)|)
 where
  integer zlevel,zfaucet,zpump,faucet,pump;
  boolean overflow,scarce;
end;
-------------------------------------------------
```

Figure 2. Bathtub model in SIGNAL.

3. Limitations of program analysis in SIGNAL

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* [2]. 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 strongly relies the automatic code generation. 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.

Example: analysis and code generation for bathtub FIG. 3 partially shows the result of the clock calculus generated automatically by the compiler. 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).

```
-------------------------------------------------
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 the clock calculus result in POLYCHRONY.

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 5.3 discusses another issue about the hierarchical control of component activations.

```
-----------------------------------------------
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 C code generated by POLYCHRONY.

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 some clock constraints in a SIGNAL program and related to the exception statements. 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, the `if` statement at line `14/14b` also contains a dead code since the variable `ghost_alarm` is always set to *false*.

4. A Boolean-interval abstraction

We define an abstraction for SIGNAL program analysis. All considered programs are supposed to be in the syntax of the core language, meaning that derived operators are replaced by their corresponding primitive statements, and there is no imbrication of operators such as in equations `06`, `08` and `13` in FIG. 2. Imbrication is broken by using fresh variables.

4.1 Notations and restrictions

Let `P` be a SIGNAL program. We denote by $X_P = \{x_1, x_2 \ldots x_n\}$ the set of all variables of `P`. We suppose that the variation interval, representing the range of possible values of each numerical signal $x_i \in X_P$, is given (see Section 5.1). 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. Our abstraction is defined for the following subset of numerical and Boolean expressions in SIGNAL statements:

$$\begin{array}{lcl} nexp & ::= & const \mid nexp \diamondsuit nexp \mid var \\ bexp & ::= & \texttt{true} \mid \texttt{false} \mid \texttt{not}\ bexp \mid var \mid bexp \\ & & \texttt{and}\ bexp \mid bexp\ \texttt{or}\ bexp \mid nexp \bowtie nexp \end{array}$$

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

We restrict ourselves to the above subset of numerical expressions because it leads to a decidable class of formulas: quantifier-free linear integer arithmetic (QF_LIA) or quantifier-free linear real arithmetic (QF_LRA). 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 $\phi(\texttt{false}) = false$,
- $\phi(b_1 \texttt{ and } b_2) = b_1 \wedge b_2$; $\phi(b_1 \texttt{ or } b_2) = b_1 \vee b_2$; $\phi(\texttt{not } b_1) = \neg b_1$,
- $\phi(n \texttt{ <= c}) = \big(\widetilde{x} \in \phi(n) \wedge \widetilde{x} \in]-\infty, c]\big)$; $\phi(n \texttt{ < c}) = \big(\widetilde{x} \in \phi(n) \wedge \widetilde{x} \in]-\infty, c[\big)$, where $x \notin X_P$ (x is a fresh variable),
- $\phi(n_1 \texttt{ <= } n_2) = \big(\widetilde{x} \in \phi(n_1 - n_2) \wedge \widetilde{x} \in [-\infty, 0]\big)$, $x \notin X_P$,
- $\phi(n_1 \diamondsuit n_2) = \widetilde{\diamondsuit}(\phi(n_1), \phi(n_2))$, an approximation of numerical operations on intervals, corresponding to $\diamondsuit$ as defined in [1]. For instance, the considered approximations on $\mathbb{N}$ are: $i \widetilde{+} j \equiv [i^- + j^-, i^+ + j^+]$ for addition and $i \widetilde{-} j \equiv [i^- - j^+, i^+ - j^-]$ where i^- and i^+ respectively denote the lower and upper bounds of an interval i.

The ϕ function is used to compute numerical and Boolean safe abstractions for our subset of expressions encountered in SIGNAL statements.

4.2 Abstraction

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 denotes the number of statements composed in `P`. In Table 2, we distinguish two possible definitions of Φ according to the type of signal `y` in each equation: (1) when `y` is of numerical type and (2) when `y` is of logical type.

Note in the abstraction of the *delay* construct, when `y` is of numerical type, a classical interval analysis would perform the convex union of the two intervals $\widetilde{c}$ and $\widetilde{x}$. Here, we avoid the approximation resulting from such a convex union by keeping the disjunction as is. By applying the above rules, the following abstractions are obtained for derived constructs for clock manipulation:

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

Example: application to the bathtub example. Let us assume that an interval-based abstract interpreter gives us the variation intervals for the integer variables (see Section 5). Then, 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:

$$\begin{array}{l} (\widehat{level} \Leftrightarrow \widehat{zlevel} \Leftrightarrow \widehat{faucet} \Leftrightarrow \widehat{pump}) \wedge (\widetilde{level} \in [1, +\infty[) \\ \wedge (\widetilde{zlevel} \in [1, +\infty[) \wedge \big(\widehat{faucet} \Leftrightarrow \widehat{zfaucet} \Leftrightarrow (\widetilde{zlevel} \in]-\infty, 4])\big) \\ \wedge (\widetilde{faucet} \in [0, +\infty[) \wedge (\widetilde{zfaucet} \in [0, +\infty[) \wedge (\widetilde{pump} \in [0, +\infty[) \\ \wedge \big(\widehat{pump} \Leftrightarrow \widehat{zpump} \Leftrightarrow (\widetilde{zlevel} \in [7, +\infty[)\big) \wedge (\widetilde{zpump} \in [0, +\infty[) \end{array}$$

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:

$$\begin{array}{l} (\widehat{overflow} \Leftrightarrow \widehat{level} \Leftrightarrow \widehat{scarce}) \wedge \big(\widetilde{overflow} \Leftrightarrow (\widetilde{level} \in [9, +\infty[)\big) \\ \wedge (\widetilde{scarce} \Leftrightarrow (\widetilde{level} \in]-\infty, 0]) \wedge (\widehat{alarm} \Leftrightarrow \widehat{scarce} \Leftrightarrow \widehat{overflow}) \\ \wedge \widehat{alarm} \Rightarrow \big(\widetilde{alarm} \Leftrightarrow (\widetilde{scarce} \vee \widetilde{overflow})\big) \\ \wedge \big(\widehat{y_2} \Leftrightarrow (\widehat{scarce} \wedge \widehat{overflow} \wedge \widetilde{scarce} \wedge \widetilde{overflow})\big) \\ \wedge (\widehat{y_2} \Rightarrow \widetilde{y_2}) \wedge (\widehat{ghost} \Leftrightarrow (\widehat{y_2} \vee \widehat{false})) \\ \wedge \Big(\widehat{ghost} \Rightarrow \Big(\big(\widehat{y_2} \wedge (\widetilde{ghost} \Leftrightarrow \widetilde{y_2})\big) \vee \big(\neg\widehat{y_2} \wedge \neg\widetilde{ghost}\big)\Big)\Big) \end{array}$$

4.3 Concretisation

Let us recall that $X = \{x_1, \ldots x_n\}$ denotes the set of all `P` variables. Intuitively, a valuation satisfying Φ captures the numerical

Table 2. Boolean-Interval abstraction of SIGNAL primitives.

process P	abstraction of P: $\Phi(P)$
`y:= R(x1,...,xn)`	$\begin{cases} \bigwedge_{i=1}^{n}(\widehat{y} \Leftrightarrow \widehat{x_i}) \wedge \left(\widehat{y} \Rightarrow \widetilde{y} \in \phi(nexp)\right) & (1) \\ \bigwedge_{i=1}^{n}(\widehat{y} \Leftrightarrow \widehat{x_i}) \wedge \left(\widehat{y} \Rightarrow (\widetilde{y} \Leftrightarrow \phi(bexp))\right) & (2) \end{cases}$ where `R(x1,...xn)` is denoted by either $nexp$ or $bexp$.
`y:= x $ 1 init c`	$\begin{cases} (\widehat{y} \Leftrightarrow \widehat{x}) \bigwedge (\widehat{y} \Rightarrow (\widetilde{y} = \widetilde{x} \vee \widetilde{y} = c)) & (1) \\ (\widehat{y} \Leftrightarrow \widehat{x}) \bigwedge (\widehat{y} \Rightarrow (\widetilde{y} \Leftrightarrow (\widetilde{c} \vee \widetilde{x}))) & (2) \end{cases}$
`y:= x when b`	$\begin{cases} (\widehat{y} \Leftrightarrow (\widehat{x} \wedge \widehat{b} \wedge \widetilde{b})) \bigwedge (\widehat{y} \Rightarrow \widetilde{y} = \widetilde{x}) & (1) \\ (\widehat{y} \Leftrightarrow (\widehat{x} \wedge \widehat{b} \wedge \widetilde{b})) \bigwedge (\widehat{y} \Rightarrow (\widetilde{y} \Leftrightarrow \widetilde{x})) & (2) \end{cases}$
`y:= x default z`	$\begin{cases} (\widehat{y} \Leftrightarrow (\widehat{x} \vee \widehat{z})) \bigwedge \left(\widehat{y} \Rightarrow \left((\widehat{x} \wedge (\widetilde{y} = \widetilde{x})) \vee (\neg\widehat{x} \wedge (\widetilde{y} = \widetilde{z}))\right)\right) & (1) \\ (\widehat{y} \Leftrightarrow (\widehat{x} \vee \widehat{z})) \bigwedge \left(\widehat{y} \Rightarrow \left((\widehat{x} \wedge (\widetilde{y} \Leftrightarrow \widetilde{x})) \vee (\neg\widehat{x} \wedge (\widetilde{y} \Leftrightarrow \widetilde{z}))\right)\right) & (2) \end{cases}$
$\mathtt{P}_1 \mid \mathtt{P}_2$	$\Phi_{P1} \wedge \Phi_{P2}$
`P where x`	$\exists\widetilde{x}, \exists\widehat{x} \,.\, \Phi_P$

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: $S_{valid}(v) = \{S \in \mathcal{E}_X | \forall i, S(i) = if\ (\widehat{x_i} = false)\ then\ \perp\ else\ \widetilde{x_i}\}$. 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 :

$$\Gamma(\Phi) \quad = \quad \{T \in \mathcal{T}_X | \forall t, T(t) \in S_{valid}(\Phi)\} \qquad (1)$$

Now, the set of constraints resulting from the abstraction of a given program P is viewed as assumptions on this program. It is used to prove properties such as clock emptiness (`x^=0`) or signal synchronization (`x1^=x2`). Note that a property is itself defined as a SIGNAL process.

We delegate the proofs to *satisfiability modulo theory* (SMT) solvers that adequately deals with Boolean and integer formulas. SMT [12] is the problem of determining whether a given first order formula Φ is satisfiable with respect to an underlying decidable first order theory. As in our case, Φ belongs to a decidable theory, SMT solvers give two kinds of answers: *sat* when the formula has a *model* (there exists a valuation that satisfies Φ) or *unsat* otherwise.

4.4 Soundness of the abstraction

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.

PROPOSITION 1. *Given a program P and a formula φ in which atoms are $\hat{x}_i$ and $\widetilde{x_i}$ ($x_i \in X_P$), if $\Phi_P \Rightarrow \varphi$, then $[\![P]\!] \subseteq \Gamma(\varphi)$. P is said to satisfy φ.* □

The proof is done thanks to the following lemma:

LEMMA 1. *For all* SIGNAL *program P, $[\![P]\!] \subseteq \Gamma(\Phi_P)$* □

PROOF 1 (Lemma 1). *By induction on the structure of P:*

- *functions/relations: $P \equiv$ `y:=f(x1,...,xn)`. We first consider the case where y is a numerical signal. Let $\widetilde{f}$ be the abstraction of the f function,* i.e., $\widetilde{f}(\widetilde{x_1}, \ldots, \widetilde{x_n}) = \phi(f(x_1, \ldots, x_n))$. *Thanks to the definition of ϕ, $\widetilde{f}$ is an over-approximation of f. Let Φ be the abstraction of P, $\Phi = \bigwedge_{i=1}^{n}(\widehat{y} \Leftrightarrow \widehat{x_i}) \wedge \widetilde{y} \in \widetilde{f}(\widetilde{x_1}, \ldots, \widetilde{x_n})$. If $v = (\widehat{\ }, \widetilde{\ })$ is a valuation satisfying Φ:*
  - *either $\forall i, \widehat{x_i} = false$ and $\widehat{y} = false$, and $\widetilde{y}$, $\widetilde{x_i}$ hold any value;*
  - *or $\forall i, \widehat{x_i} = true$ and $\widehat{y} = true$ and $\widetilde{y} \in \widetilde{f}(\widetilde{x_1} \ldots, \widetilde{x_n})$;*

  *$S_{valid}(\Phi)$ is the set of all valuations of the previous form. Now, given a trace T of $[\![P]\!]$ and $t_0 \in \mathbb{N}$, either $\forall i\ T(t_0)(y) = T(t_0)(x_i) = \perp$ or $T(t_0)(y) = f(T(t_0)(x_1), \ldots, T(t_0)(x_n)) \in \widetilde{f}(T(t_0)(x_1), \ldots, T(t_0)(x_n))$ (*SIGNAL *semantics and over-approximation of f), which means in both cases that $T \in \Gamma(\Phi_P)$.*
  When y is a Boolean signal, $[\![P]\!] \subseteq \Gamma(\Phi_P)$ is similarly proved.
- *for the delay, under-sampling and merging constructs, the same proof sketch holds. It is not detailed here due to lack of space.*
- *composition: $P \equiv P1|P2$. We have $[\![P]\!] \subseteq [\![P1]\!] \subseteq \Gamma(\Phi_{P1})$ by applying the induction hypothesis. In a similar way, we also have $[\![P]\!] \subseteq \Gamma(\Phi_{P2})$. Then, $[\![P]\!] \subseteq \Gamma(\Phi_{P1}) \cap \Gamma(\Phi_{P2})$. Since $\Gamma(\Phi_{P1}) \cap \Gamma(\Phi_{P2}) \subseteq \Gamma(\Phi_{P1} \wedge \Phi_{P2})$, we have $[\![P]\!] \subseteq \Gamma(\Phi_{P1} \wedge \Phi_{P2}) = \Gamma(\Phi_{P1|P2}) = \Gamma(\Phi_P)$.*
- *restriction: $P \equiv$ `P1 where x`. By definition, we have $[\![P]\!] \subseteq [\![P1]\!]$ on $X_{P1-\{x\}}$. On the other hand, by induction $[\![P1]\!] \subseteq \Gamma(\Phi_{P1})$. Since $\Gamma(\Phi_{P1}) \subseteq \Gamma(\exists\widehat{x}, \exists\widetilde{x} \,.\, \Phi_{P1})$, we obtain $[\![P]\!] \subseteq \Gamma(\Phi_P)$.*

Now, we prove Proposition 1:

PROOF 2. *Let $T \in [\![P]\!]$. According to Lemma 1, $T \in \Gamma(\Phi_P)$, which means $\forall t, T(t) \in S_{valid}(\Phi_P)$ (Formula (1) defining the concretisation). As $\Phi_P \Rightarrow \varphi$, every valuation v satisfying Φ_P also satisfies φ. Any event S of $S_{valid}(\Phi_P)$ belongs to $S_{valid}(\varphi)$, hence $\forall t, T(t) \in S_{valid}(\varphi)$. Finally, we have $T \in \Gamma(\varphi)$ (Formula (1) again).*

5. Implementation

We present the different steps of our approach, and the tools we use to implement it. Then, we illustrate them on `Bathtub`.

5.1 Analysis flow implementation

From a global point of view, our approach takes a program P as input. Different tools are combined to achieve a suitable abstraction of P and to prove properties, mainly those which are not addressed by the SIGNAL compiler. Finally, the satisfied properties are made explicit in P so that the compiler can exploit them for a more precise static analysis and efficient code generation. FIG. 5 summarizes the different steps.

1. **Pre-computation of variation intervals.** This step aims at computing the variation intervals for all numerical variables of a given SIGNAL program. For input signals, the corresponding

A. Gamatié and Gonnord , accepted to ACM Languages, Compilers, Tools and Theory for embedded Systems (LCTES 2011, Chicago) ---------------------- *©2011 ACM*

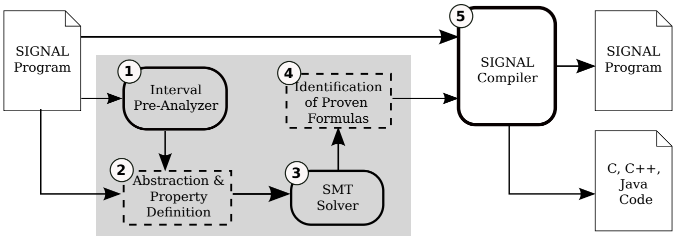


Figure 5. An overview of our approach: steps and tools.

intervals are assumed to be known. First, the program is compiled into a counter automaton in which all Boolean values are abstracted. This step is an adaptation of the algorithm presented in [18] and is very similar to the compilation of LUSTRE programs into OC [21],[22]. Afterwards, we apply the classical abstract interpretation on the interval lattice [10] to compute an over-approximation of the variation interval of each numerical signal. For that, we use the INTERPROC tool [24], which implements this technique.

2. **Abstraction and property generation.** The input program is translated according to the abstraction definition provided in Section 4. In addition, clock synchronization and clock emptiness properties are produced according to the program variables and translated as well. For the moment, this step is performed manually, even though its implementation is straightforward.
3. **SMT-based analysis.** We delegate the proof of the above properties against the program abstraction to an SMT solver that adequately deal with Boolean and numerical formulas. As the considered formulas belong to decidable theories, this solver gives two kinds of answers: *sat* when the formula has a *model* (there exists a valuation that satisfies it); or *unsat* otherwise.

   The formulas obtained from the previous step are encoded in the SMTLIB common format [3], as an input for SMT solvers. For our example, we consider the Yices [13] solver, which is one of the best two solvers dealing with unquantified linear integer arithmetic in the last SMTCOMP competition([30]).

   The output of Yices determined the result of the applied static analysis. This result can be exploited for a better code generation, as addressed by the next step.
4. **Concretisation of proven formulas for composition with the program.** For all proven clock properties that the compiler is not able to address, we consider a possible SIGNAL program that belongs to their concretisation. Then we compose this program with the initial program, *without changing its semantics* [26]. The result of the composition is to be compiled.
5. **Clock analysis and code generation.** The result of the previous step is a program semantically equivalent to the initial one. The advantage is that the compiler can exploit it in a more adequate way so as to permit the proof of intricate clock properties involving numerical expressions. This has a direct and strong impact on the quality of the code generated by the compiler.

5.2 Application 1: clock equivalence, exclusion and emptiness

Let us illustrate the previous analysis flow on `Bathtub` (FIG. 2).

In the first step, we only focus on the subset of statements which are defined between the lines 04 and 09. These statements cover the definition of all numerical signals of the program. Their compilation provides the automaton shown in FIG.6 (x' means the value of x after assignment). After interval analysis on this automaton, we get: $level, zlevel \in [1, +\infty]$, $faucet, zfaucet \in [0, \infty]$, $pump, zpump \in [0, +\infty]$.

The second step of our method then computes the abstraction of the `Bathtub` program thanks to the information computed in step 1. The obtained formula $\Phi_{\texttt{Bathtub}}$ is the one already provided in Section 4.2. Furthermore, we have to generate clock synchronization and clock emptiness properties (formula φ) from the program. Among these properties, let us focus on 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: $\neg(\widetilde{scarce} \wedge \widetilde{overflow} \wedge \widehat{scarce} \wedge \widehat{overflow})$
3. `alarm` and `level` have the same clock: $\widehat{alarm} \Leftrightarrow \widehat{level}$

In the third step, we encode the formula $\Phi_{\texttt{Bathtub}} \wedge \neg\varphi$, where φ denotes the property to be checked. For instance, to check whether or not the signals `alarm` and `level` are synchronous, we use $\varphi = \widehat{alarm} \Leftrightarrow \widehat{level}$. With the Yices SMT-solver, we get *unsat*, which means that $\Phi_{\texttt{Bathtub}} \models \varphi$. Thanks to Proposition 1, the property φ is satisfied by `Bathtub`. Here, the previous three formulas are proven.

In the fourth step, the program `Bathtub` is composed with programs that belong to the set of concretisations of φ. For property "the water cannot overflow and be scarce at the same time", a possible concretisation is the program:

```
true when scarce when overflow^=^0.
```

Finally, we obtain the process in FIG. 7.

```
-------------------------------------------------------
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: where
16:  integer zlevel,zfaucet,zpump,faucet,pump;
17:  boolean overflow,scarce;
18:end;
-------------------------------------------------------
```

Figure 7. Bathtub model composed with one clock constraint.

In step 5, the compiler can now exploit the proven properties for an enhanced clock calculus and code generation. For the program of FIG. 7, the compiler is able to infer that the value of the `ghost_alarm` signal is always equal to `false`, as illustrated in FIG. 8. This is represented at lines 08 and 09.

```
------------------------------------------------------
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_ghost_alarm ^= ghost_alarm
09:    | (| ghost_alarm := not CLK_ghost_alarm |)
10:    |) ... |)
------------------------------------------------------
```

Figure 8. A sketch of the clock calculus for `Bathtub_Bis`.

$$t_4 \begin{cases} pump' = zpump + 1, \\ faucet' = faucet \end{cases}$$
$$t_3 \begin{cases} pump' = zpump, \\ faucet' = zfaucet + 1 \end{cases}$$
$$t_2 \begin{cases} pump' = zpump, \\ faucet' = faucet \end{cases}$$
$$t_1 \begin{cases} level' = zlevel + faucet - pump, \\ zlevel' = level, zfaucet' = faucet, \\ zpump' = pump \end{cases}$$
$$t_0 \begin{cases} zlevel' = 1, level' = 1, \\ zfaucet' = 0, faucet' = 1, \\ zpump' = 0, pump' = 0 \end{cases}$$

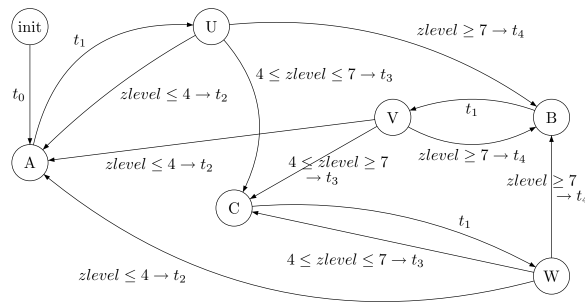


Figure 6. Bathtub integer kernel, after compilation.

Nevertheless, there still exist some unsolved clock constraints in FIG. 8. Returning to step 4, we compose the `Bathtub` process with all the proven clock properties concretised by the following programs:

- `faucet ^* pump ^= ^0`
- `true when scarce when overflow ^= ^0`
- `alarm ^= level`

Then, we obtain the process `Bathtub_Ter` shown in FIG. 9. The result of its analysis performed by the compiler is in FIG. 10.

```
-----------------------------------------------------
01:process Bathtub_Ter =
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;
-----------------------------------------------------
```

Figure 9. Bathtub model composed with all clock constraints.

```
---------------------------------------------------
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%
---------------------------------------------------
```

Figure 10. A sketch of the clock calculus for `Bathtub_Ter`.

The whole set of constraints inferred by the compiler is now restricted to the only 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 (line 04/04b in FIG. 10). Finally, the corresponding generated code is provided in FIG. 11, where the dead code is avoided.

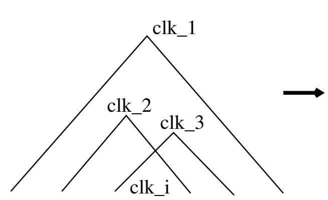
```
--------------------------------------------------
01:   { ghost_alarm = FALSE;
02:    /* produce output value
03:         for the signal ghost_alarm */ } ...
--------------------------------------------------
```

Figure 11. A sketch of the C code for `Bathtub_Ter`.

5.3 Application 2: control structure optimization

Our abstraction is also usable for optimizing the control structure of of the code generated by the SIGNAL compiler. As discussed in Section 3, 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 different actions described in a program, they are translated as conditional statements in generated code, for instance 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. 12: 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. Notice 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.




```
if (clk_1)
  { ...;
    if (clk_2)
      { ... };
    if (clk_3)
      { ... };
    if (clk_2 && clk_3)
      { if (clk_i)
          { ... };
          ...;
      }...;
  }
```

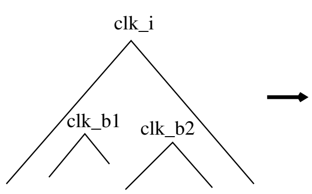
Figure 12. Clock hierarchy-based code generation.

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 program defined in FIG. 13, where two signals `b1` and `b2` of `event` type, are defined according to the value of an integer signal `i`. The event `b1` occurs when the value of `i` is between -10 and 10, while `b2` occurs when `i` is between -5 and 5. One can straightforwardly deduce that the clock of `b2` is a subset of the clock of `b1`.

```
------------------------------------------------------
process Inclusion =
  ( ? integer i;
    ! event b1, b2;)
  (| b1 := when ((i<10) and (i>-10))
   | b2 := when ((i<5) and (i>-5))
   |);
------------------------------------------------------
```

Figure 13. Bathtub model composed with one clock constraint.

The clock hierarchy computed by the compiler is depicted in FIG. 14. While the clocks of `b1` and `b2` appear to be sub-clocks of that the clock of `i`, the clock hierarchy between `b1` and `b2` is not reflected. This leads to a control structure in generated code where the trigger testing related to `b2` is always performed, even though that of `b1` is false while it is unnecessary.



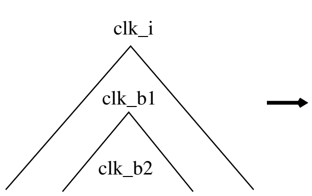

```
if (clk_i)
  { ...;
   if (clk_b1)
     { ... };
   if (clk_b2)
     { ... };
  }
```

Figure 14. Clock hierarchy for `Inclusion` process.

Actually the above situation can be avoided for an optimized code by enabling a finer analysis (not with a Boolean abstraction only) of the numerical expressions defining the clocks of `b1` and `b2`. This can be solved easily by considering our abstraction. The formula to be proven (fourth step in the proposed flow) encodes the clock inclusion of `b1` and `b2` (in SIGNAL, `b1 ^* b2 ^= ^b2`):

$$\begin{array}{ll} (\widehat{b_1} \Leftrightarrow (i > -10 \wedge i < 10)) & \wedge \\ (\widehat{b_1} \Leftrightarrow (i > -5 \wedge i < 5)) & \wedge \\ ((\widehat{b_1} \wedge \widehat{b_2}) \Leftrightarrow \widehat{b_2}) & \end{array}$$

By verifying this formula, the clock hierarchy can be modified in the compiler as shown in FIG. 15, from which an optimized code is generated.




```
if (clk_i)
  { ...;
   if (clk_b1)
     { ...;
      if (clk_b2)
        { ... };
     }
  }
```

Figure 15. Optimized clock hierarchy for `Inclusion` process.

The previous examples (Applications 5.1 and 5.3 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 consists 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[11]. 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 [14] by identifying *regions* in a control flow graph.

6. Related work

In [16, 17], 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 a *non intrusive* solution that consists in deducing additional information from an initial program specification with SMT solvers. This therefore enables the compiler to have an explicit and rich set of constraints for a better program analysis and code analysis by using its current clock calculus technique.

Another tentative of combining numerical and Boolean techniques has been done for LUSTRE verification. In [23], the technique used is a dynamic partitioning of the control flow obtained by LUSTRE compilation (which contains a few number of control points) with respect to some constraints coming from the proof goal. Conversely, our approach is not dependent on a proof goal, and the Boolean variables are not hidden in the control (except for the step 1). In addition, LUSTRE compilation [21] suffers from the same lack of precision concerning numerical variables. Indeed, no numerical analysis is done during compilation. Hence, our method could be considered for improvement.

In [19], SMT is used to verify safety properties on LUSTRE programs. 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 multi-clocked specifications. Neither an SMT solver nor the LUSTRE compiler makes this analysis possible.

An important work is the polyhedral-based static analysis for synchronous languages of [6]. The authors give a technique based on fix-point iteration on a lattice combining Boolean and affine constraints. Our technique is less precise because it only uses interval approximation. However, the complexity in our case is lesser and the implementation is much simpler.

Finally, a relevant study presented in [29], concerns the definition of a clock language $\mathcal{CL}$ aiming to capture the static control part of SIGNAL programs. 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 covers all SIGNAL programs and offers more expressivity than $\mathcal{CL}$.

7. Conclusion

In this paper, we presented a combination of the synchronous approach with SMT solving for a powerful static analysis of embedded system specifications. We considered the SIGNAL language for behavior description. The analysis achieved by its compiler, which is based on Boolean abstraction, has been extended in our approach by defining a more expressive mixed Boolean-interval 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. In addition, their related properties are extensively exploited by the SIGNAL compiler for optimizing the automatic code generation process. We showed, in a pragmatic way, how the new abstraction combined with SMT solving and interval abstract interpretation techniques infers useful information, which strongly help the compiler to solve more clock constraints and generate higher quality code, *e.g.*, by avoiding dead code. A prototype tool-chain has been proposed for this purpose.

Among the perspectives to this work, we mention the enhancement of the current prototype tool-chain by making it fully automatic. This will be validated on more case studies. We will investigate the automatic generation of the proof goals, as this step needs for the moment the developer expertise.

Acknowledgments

The authors wish to thank the anonymous referees for their useful comments on this paper. They also would like to thank Jan Reineke for his valuable suggestions to improve the contents of this paper.

References

[1] G. Alefeld and J. Hertzberger. *Introduction to Interval Computation.* Academic Press, NY, 1983.

[2] T. Amagbegnon, L. Besnard, and P. Le Guernic. Arborescent canonical form of Boolean expressions. Technical Report 2290, INRIA, June 1994. URL `www.inria.fr/rrrt/rr-2290.html`.

[3] C. Barrett, S. Ranise, A. Stump, and C. Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). www.SMT-LIB.org, 2008.

[4] 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.

[5] G. Berry. The foundations of ESTEREL. In *Proof, Language and Interaction: Essays in Honour of Robin Milner*. MIT Press, 2000.

[6] 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*, pages 51–68. Springer-Verlag, September 1999.

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

[8] R. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE transactions on computers*, C-35(8):677–691, August 1986.

[9] D. Chapiro. *Globally Asynchronous Locally Synchronous Systems*. PhD thesis, Stanford University, 1984.

[10] P. Cousot and R. Cousot. Static determination of dynamic properties of programs. In *Proceedings of the Second International Symposium on Programming*, pages 106–130. Dunod, Paris, France, 1976.

[11] 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.*, 13:451–490, October 1991. ISSN 0164-0925.

[12] L. de Moura and N. Bjorner. Satisfiability Modulo Theories: An Appetizer. In *Brazilian Symposium on Formal Methods (SBMF'2009), Gramado, Brazil*, August 2009.

[13] B. Dutertre and L. de Moura. Yices sat-solver. `http://yices.csl.sri.com/`, 2009.

[14] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9:319–349, July 1987. ISSN 0164-0925.

[15] A. Gamatié. *Designing Embedded Systems with the* SIGNAL *Programming Language: Synchronous, Reactive Specification*. Springer, New York, 2009. ISBN 978-1-4419-0940-4.

[16] 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.

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

[18] L. Gonnord and N. Halbwachs. Abstract acceleration to improve precision of linear relation analysis. Research report, Verimag, 03 2010. URL `http://laure.gonnord.org/pro/papers/rr-verimag10.pdf`.

[19] 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*, pages 1–9, Piscataway, NJ, USA, 2008. IEEE Press. ISBN 978-1-4244-2735-2.

[20] 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.

[21] 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*, 18(9):785–793, September 1992.

[22] N. Halbwachs, Y.-E. Proy, and P. Roumanoff. Verification of real-time systems using linear relation analysis. *Formal Methods in System Design: An International Journal*, 11(2):157–185, August 1997.

[23] B. Jeannet. Dynamic partitioning in linear relation analysis. application to the verification of reactive systems. *Formal Methods in System Design*, 23(1):5–37, July 2003.

[24] G. Lalire, M. Argoud, and B. Jeannet. Interproc : an interprocedural analyzer for imparative languages. `http://pop-art.inrialpes.fr/people/bjeannet/bjeannet-forge/interproc`, 2009.

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

[26] P. Le Guernic, J.-P. Talpin, and J.-C. Le Lann. Polychrony for System Design. *Journal for Circuits, Systems and Computers*, 12(3):261–304, April 2003.

[27] G. Magklis, G. Semeraro, D. Albonesi, S. Dropsho, S. Dwarkadas, and M. Scott. Dynamic frequency and voltage scaling for a multiple-clock-domain microprocessor. *Micro, IEEE*, 23(6):62 – 68, November 2003. ISSN 0272-1732. doi: 10.1109/MM.2003.1261388.

[28] J. Muttersbach, T. Villiger, and W. Fichtner. Practical design of globally asychronous locally synchronous systems. In *International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC'00)*, pages 52–61, 2000.

[29] M. Nebut. Specification and analysis of synchronous reactions. *Formal Aspects of Computing*, 16(3):263–291, august 2004.

[30] A. Stump and M. Deters. The SMT-COMP 2009 Website, 2009. `http://www.smtcomp.org/2009`.