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.](image-url)

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...
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 the 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 (escape) 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, a 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 discuss 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 formula. 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 $\bot$ 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 instant at which a signal $x$ is present is referred to as its clock, noted $\text{clk}$. 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:** $y := R(x_1, \ldots, x_n)$ where $y$, $x_1$, ..., $x_n$ are signals and $R$ is a point-wise n-ary relation/function extended canonically to signals. This construct imposes $y$, $x_1$, ..., $x_n$ $i)$ to be simultaneously present, i.e. $y = ^*x_1 = \ldots = ^*x_n$ (i.e. synchronous signals), and $ii)$ to hold values satisfying $y := R(x_1, \ldots, x_n)$ whenever they occur.

- **Delay:** $y := x \% 1 \text{ init } c$ 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:** $y := x \% b$ 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 $[b] \cup [-b] = ^*b$ and $[b] \cap [-b] = \emptyset$), while $ii)$ $y$ holds
Table 1. Trace semantics for SIGNAL primitives.

<table>
<thead>
<tr>
<th>process P</th>
<th>semantics of P: [P]</th>
</tr>
</thead>
<tbody>
<tr>
<td>y := R(x₁,...,xₙ)</td>
<td>{ T ∈ Tₚ [x₁,...,xₙ,y] / ∀t ∈ N, (∀i, T(t)(xᵢ) = T(t)(y) = ⊥) or (T(t)(y) = ⊥ ∧ ∀i, T(t)(xᵢ) ≠ ⊥ ∧ T(t)(y) = R(T(t)(x₁),...,T(t)(xₙ))) }</td>
</tr>
<tr>
<td>y := x $ 1 init c</td>
<td>{ T ∈ T₂ [x,y ] / ∀t ∈ N, (T(t)(x) = T(t)(y) = ⊥) or (T(t)(y) ≠ ⊥ ∧ T(t)(x) ≠ ⊥ ∧ T(t)(y) = c and (t ≥ t₀) ⇒ (∃i, t = tᵢ, T(tᵢ+1)(y) = T(t)(x)) } with t₀ = inf{t</td>
</tr>
<tr>
<td>y := x when b</td>
<td>{ T ∈ T₂ [x,y ] / ∀t ∈ N, (T(t)(y) = ⊥ ∧ T(t)(x) = T(t)(y)) }</td>
</tr>
<tr>
<td>y := x default y</td>
<td>{ T ∈ T₂ [x,y ] / ∀t ∈ N, (T(t)(y) = ⊥ ∧ T(t)(x) = T(t)(y)) }</td>
</tr>
<tr>
<td>z := x default y</td>
<td>{ T ∈ T₂ [x,y ] / ∀t ∈ N, (T(t)(y) = ⊥ ∧ T(t)(x) = T(t)(y)) }</td>
</tr>
<tr>
<td>P₁ ∣ P₂</td>
<td>Assuming that [P₁] ⊆ Tₚ₁, [P₂] ⊆ Tₚ₂, { T ∈ T₂ [x,y ] / X₁,T ∈ [P₁] ∧ X₂,T ∈ [P₂] }</td>
</tr>
<tr>
<td>P₁ where x</td>
<td>Assuming that [P₁] ⊆ Tₚ₁, { T ∈ T₂ [x ] / ∃T₁ ∈ [P₁], [X₁ - {x}], T₁ = T }</td>
</tr>
</tbody>
</table>

the value of x at those logical instants. The sub-clock [b] (resp. [¬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 0 to z when either x or y are present, i.e. z = x ∪ y, while ii) z holds the value of x uppermost, otherwise that of y.

- **Composition:** P ⊨ 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.

- **Composition (or Hiding):** P ⊨ P₁ where x, where P₁ and x are 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.

**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 where b is a Boolean signal.

- **Restriction (or Hiding):** P ⊨ P₁ where x, where P₁ and x are 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.

**Definition 1 (events).** Given a non-empty set X₁ ⊆ X, the set of events on X₁, denoted by Eₓ₁, is the set of all applications (functions) m defined from X₁ to DX₁.

The expression m(x) = ⊥ means x holds no value while m(x) = v means x holds the value v, and m(X₁) = \{m(x) | x ∈ X₁\}. The set of events on X₁ is denoted by Eₓ₁ = X₁ → DX₁, and the set of all possible events is therefore E = \bigcupₓ₁∃ₓ₁ Eₓ₁. By convention, the event on an empty set of ports is represented by E₀ = ∅.

**Definition 2 (traces).** Given a non-empty set X₁ ⊆ X, the set of traces on X₁, denoted by Tₓ₁ := X₁ → Eₓ₁, is the set of all applications T defined from the set N of natural numbers to Eₓ₁.

The set of all possible traces is T⁺ = \bigcupₓ₁∃ₓ₁ Tₓ₁. Moreover, T₀ = 1 = N → E₀.

**Definition 3 (trace restriction).** Given a non-empty set X₁ ⊆ X, and a set X₂ ⊆ X₁ with a trace T being defined on X₁, the restriction of T(t) to X₂, noted X₂,T : N → Eₓ₂, satisfies: ∀t ∈ N, ∀x ∈ X₂, Tₓ₂(T(t)(x)) = T(t)(x).

We have 0.T ∈ T₀ (which is a singleton).

We also define the trace restriction (or projection) of set of traces T to X ⊆ X₁ as follows: X.T = \{T | T ∈ T \}.

A process on a set of variables X₁ ⊆ X is a set of constrained traces on X₁. In other words, it is a subset of Tₓ₁. The semantics of statements defining a process P is denoted by a set of traces [P]. Each SIGNAL primitive construct defines an elementary process whose trace semantics is given in Table 1.

### 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

\[ y := \text{R}(x₁,...,xₙ) \]

\[ (T(t)(y) = \bot \land \forall i, T(t)(xᵢ) = \bot) \text{ or } (T(t)(xᵢ) = \bot \land \forall i, T(t)(y) = \bot) \]
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 the static analysis of SIGNAL. This is necessary in order to prove the consistency of clock relations and 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 $[-b]$, which are referred to as condition-clocks. If b is defined by a numerical expression such as an integer comparison, $[b]$ and $[-b]$ 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: if (C_level) 02: { C_zfaucet = level <= 4; 03: C_zpump = level >= 7; 04: if ((C_zpump) != (C_level)) 05: polychrony_exception("..."); 06: if (C_zfaucet) != (C_level) 07: polychrony_exception("..."); 08: if (C_zfaucet) { faucet = zfaucet + 1; } 09: if (C_zpump) { pump = zpump + 1; } 10: level = (level + faucet) - pump; 11: overflow = level >= 9; 12: alarm = scarce || overflow; 13: process level, alarm, ghost_alarm; 14: { if (C_ghost_alarm) 15: /*production of level and alarm*/ 16: C_106 = overflow && scarce; 17: C_109 = (C_level ? C_106 : FALSE); 18: if (C_ghost_alarm) 19: if (C_109) ghost_alarm = TRUE; 19b: else ghost_alarm = FALSE; 20: /*production of ghost_alarm*/ ...

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
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: \( \bar{x}_i \) and \( \bar{x}_i \), encoding respectively its clock and values.

The abstract semantics of the program, is a set of couples of the form \((\bar{\cdot}, \bar{\cdot})\) where:

- \( \text{function } \bar{\cdot} : X_P \rightarrow \mathbb{B} = \{\text{true, false}\} \) assigns to a variable a Boolean value;
- \( \text{function } \bar{\cdot} : X_P \rightarrow \mathbb{R} \cup \mathbb{B} \) assigns to a variable a numerical or Boolean value.

This abstract set is represented as a first order logic formula \( \Phi_P \), in which atoms are \( \bar{x}_i \) and \( \bar{x}_i \), 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:

\[
\begin{align*}
\text{next} &::= \text{const} \mid \text{next} \land \text{next} \mid \text{var} \\
\text{bexp} &::= \text{true} \mid \text{false} \mid \text{not bexp} \mid \text{var} \mid \text{bexp} \\
&\mid \text{bexp} \lor \text{bexp} \mid \text{next} \land \text{next}
\end{align*}
\]

where the symbols \( \text{const} \) and \( \text{var} \) respectively denote a constant and a signal variable \((x, y, \ldots) : \in \{<, >, =, \neq, \neq\} \) and \( \diamond \in \{+, *, -, /\}\).

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 \( \phi \) for these expressions by induction on their structure as follows:

- atoms: given a signal \( x \), if \( x \) is of Boolean or numeric type, \( \phi(x) = \bar{x} \), if \( x \) is of event type, \( \phi(x) = \text{true} \).
- \( \phi(\text{true}) = \text{true} \) and \( \phi(\text{false}) = \text{false} \).
- \( \phi(b_1 \land b_2) = \bar{b}_1 \land \bar{b}_2; \phi(b_1 \lor b_2) = \bar{b}_1 \lor \bar{b}_2; \phi(\text{not} b_1) = \neg \bar{b}_1 \).
- \( \phi(n \leq c) = (\bar{x} \in \phi(n) \land \bar{x} \in [-\infty, c]); \phi(n < c) = (\bar{x} \in \phi(n) \land \bar{x} \in [-\infty, c]) \), where \( x \in X_P \) (x is a fresh variable),
- \( \phi(n_1 \land n_2) = (\bar{x} \in \phi(n_1) \land \bar{x} \in [-\infty, 0]) \), \( x \notin X_P \),
- \( \phi(n_1 \lor n_2) = (\bar{x} \in \phi(n_1 \lor n_2) \land \bar{x} \in [-\infty, 0]) \).

4.2 Abstraction

We define \( \Phi_P \) as the intersection of the abstractions of statements \( \text{stmt}_i \) of P:

\[
\Phi_P = \bigwedge_{i=1}^{n} \Phi(\text{stmt}_i)
\]

where \( n \) denotes the number of statements composed in P. In Table 2, we distinguish two possible definitions of \( \Phi \) 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 \( \bar{x} \) and \( \bar{x} \). 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:

- \( \Phi(y := x_1 \ast x_2) = (\bar{y} \leftrightarrow (\bar{x}_1 \lor \bar{x}_2) \land (\bar{y} \Rightarrow \bar{y})) \).
- \( \Phi(y := x_1 \ast x_2) = (\bar{y} \leftrightarrow (\bar{x}_1 \land \bar{x}_2)) \land (\bar{y} \Rightarrow \bar{y}) \).
- \( \Phi(y := x_1 \ast x_2) = (\bar{y} \leftrightarrow (\bar{x}_1 \land \bar{x}_2)) \land (\bar{y} \Rightarrow \bar{y}) \).

**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 \( \Phi_{\text{bathtub}} = \Phi_{P_1} \land \Phi_{P_2} \), where \( \Phi_{P_1} \) equals to:

\[
\begin{align*}
\text{level} \Rightarrow \text{zlevel} &\Rightarrow \text{faucet} \Rightarrow \text{pump} \land (\text{level} \in [1, +\infty]) \\
&\land (\text{zlevel} \in [1, +\infty]) \land (\text{faucet} \Rightarrow \text{zfaucet} \Rightarrow (\text{zlevel} \in [-4, \infty])) \\
&\land (\text{faucet} \in [0, +\infty]) \land (\text{zfaucet} \in [0, +\infty]) \land (\text{pump} \in [0, +\infty]) \\
&\land (\text{pump} \Rightarrow \text{zpump} \Rightarrow (\text{zlevel} \in [7, +\infty]) \land (\text{zpump} \in [0, +\infty]))
\end{align*}
\]

For \( \Phi_{P_2} \), we first rewrite equation at line 13/14 as follows:

\[
(| y1 := \text{true when scarce} \mid y2 := y1 \text{ when overflow} \mid \text{ghost_alarm} := y2 \text{ default false} |)
\]

Then, we obtain that \( \Phi_{P_2} \) equals to:

\[
\begin{align*}
\text{overflow} \Rightarrow \text{level} \Rightarrow \text{scarce} \land (\text{overflow} \Rightarrow (\text{level} \in [9, +\infty]) \\
&\land (\text{scarce} \Rightarrow (\text{level} \in [-\infty, 0]) \land (\text{alarm} \Rightarrow \text{scarce} \Rightarrow \text{overflow}) \\
&\land (\text{alarm} \Rightarrow (\text{scarce} \Rightarrow \text{overflow}) \\
&\land (\text{ghost} \Rightarrow (\text{y2} \land (\text{ghost} \Rightarrow \text{y2} \lor \text{false}))))
\end{align*}
\]

4.3 Concretisation

Let us recall that \( X = \{x_1, \ldots x_n\} \) denotes the set of all P variables. Intuitively, a valuation satisfying \( \Phi \) captures the numerical
and Boolean values of signals at a given logical instant. Given a valuation \( v = (\sim, \sim) \), where all variables have been assigned some values, we first construct a set of events whose values are assigned accordingly: \( S_{valid}(v) = \{ S \in \Sigma \mid v, S(i) = \text{true} \} \). The set of all “valid” events is defined as \( S_{valid}(\Phi) = \bigcup_{\psi \in \Phi} S_{valid}(\psi) \). Finally, the concretisation of \( \Phi \) is the set of traces whose instantaneous values always verify \( \Phi \):

\[
\Gamma(\Phi) = \{ T \mid T(x) \in S_{valid}(\Phi) \}
\]

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 (\( x_1^* = x_2^* \)). 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 \( \Phi \) is satisifiable with respect to an underlying decidable first order theory. As in our case, \( \Phi \) 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 \( \Phi \)) 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 \( \varphi \) in which atoms are \( x_i \) and \( \bar{x}_i \) (\( x_i \in X_P \)), if \( \Phi_P \models \varphi \), then \( \llbracket P \rrbracket \subseteq \Gamma(\varphi) \). \( P \) is said to satisfy \( \varphi \).

The proof is done thanks to the following lemma:

**Lemma 1.** For all SIGNAL program \( P \), \( \llbracket P \rrbracket \subseteq \Gamma(\Phi_P) \).

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

- **functions/relations:** \( P \equiv y := f(x_1, \ldots, x_n) \). We first consider the case where \( y \) is a numerical variable. Let \( \bar{f} \) be the abstraction of the \( f \) function, i.e., \( \bar{f}(\bar{x}_1, \ldots, \bar{x}_n) = f(x_1, \ldots, x_n) \).

- Thanks to the definition of \( \bar{f} \), each \( \bar{f}(\bar{x}_1, \ldots, \bar{x}_n) \) is a \( f \) over-approximation of \( f \). Let \( \Phi \) be the abstraction of \( P \), \( \Phi = \bigwedge_{i=1}^{n} (\bar{y} \equiv \bar{x}_i) \land \bar{y} \in \bar{f}(\bar{x}_1, \ldots, \bar{x}_n) \) if \( v = (\sim, \sim) \) is a valuation satisfying \( \Phi \):
  - **either:** \( \forall i, \bar{x}_i = \text{false and } \bar{y} = \text{false} \), and \( y, \bar{x}_i \) hold any value;
  - **or:** \( \forall i, \bar{x}_i = \text{true and } \bar{y} = \text{true and } \bar{y} \in \bar{f}(\bar{x}_1, \ldots, \bar{x}_n) \);

\( S_{valid}(\Phi) \) is the set of all valuations of the previous form. Now, given a trace \( T \) of \( \llbracket P \rrbracket \) and \( t_0 \in \mathbb{N} \), either \( \forall i, T(t_0)(y) = T(t_0)(x_i) = \bot \) or \( T(t_0)(y) = f(T(t_0)(x_1), \ldots, T(t_0)(x_n)) \in \bar{f}(T(t_0)(x_1), \ldots, T(t_0)(x_n)) \). (Signal semantics and over-approximation of \( f \), which means in both cases that \( T \in \Gamma(\Phi) \).

When \( y \) is a Boolean signal, \( \llbracket P \rrbracket \subseteq \Gamma(\Phi_P) \) 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:** \( P = P_1 \parallel P_2 \). We have \( \llbracket P \rrbracket \subseteq \llbracket P_1 \rrbracket \parallel \llbracket P_2 \rrbracket \) by applying the induction hypothesis. In a similar way, we also have \( \llbracket P \rrbracket \subseteq \Gamma(\Phi_P) \). Then, \( \llbracket P \rrbracket \subseteq \Gamma(\Phi_P) \parallel \Gamma(\Phi_P) \), hence \( \llbracket P \rrbracket \subseteq \Gamma(\Phi_P) \parallel \Gamma(\Phi_P) \).

Since \( \Gamma(\Phi_P) \parallel \Gamma(\Phi_P) \subseteq \Gamma(\Phi_{P_1} \land \Phi_{P_2}) \), we have \( \llbracket P \rrbracket \subseteq \Gamma(\Phi_{P_1} \land \Phi_{P_2}) = \Gamma(\Phi_P) \).

- **restriction:** \( P \equiv P_1 \text{ where } x \). By definition, we have \( \llbracket P_1 \rrbracket \subseteq \llbracket P \rrbracket \). On the other hand, by induction \( \llbracket P_1 \rrbracket \subseteq \Gamma(\Phi_{P_1}) \). Since \( \Gamma(\Phi_P) \subseteq \Gamma(\exists x, \exists x \cdot \Phi_P) \), we obtain \( \llbracket P \rrbracket \subseteq \Gamma(\Phi_P) \).

Now, we prove Proposition 1:

**Proof 2.** Let \( T \in \llbracket P \rrbracket \). According to Lemma 1, \( T(t) \in \Gamma(\Phi_P) \), which means \( \forall t, T(t) \in S_{valid}(\Phi_P) \). (Formula 1 defining the concretisation). As \( \Phi_P \models \varphi \), every valuation \( v \) satisfying \( \Phi_P \) also satisfies \( \varphi \). Any event \( S \in S_{valid}(\Phi_P) \) belongs to \( S_{valid}(\varphi) \), hence \( v, T(t) \in S_{valid}(\varphi) \). Finally, we have \( T(t) \in \Gamma(\varphi) \) (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

---

**Table 2.** Boolean-Interval abstraction of SIGNAL primitives.

<table>
<thead>
<tr>
<th>process ( P )</th>
<th>abstraction of ( P : \Phi(P) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( y := R(x_1, \ldots, x_n) )</td>
<td>( \bigwedge_{i=1}^{n} (\bar{y} \equiv \bar{x}<em>i) \land (\bar{y} \Rightarrow (\bar{y} \equiv \phi(\text{nezexp})) ) (1) ( \bigwedge</em>{i=1}^{n} (\bar{y} \equiv \bar{x}_i) \land (\bar{y} \Rightarrow (\bar{y} \equiv \phi(\text{bezexp})) ) (2) where ( R(x_1, \ldots, x_n) ) is denoted by either ( \text{nezexp} ) or ( \text{bezexp} ).</td>
</tr>
<tr>
<td>( y := x \ $ \ 1 \ \text{init} \ c )</td>
<td>( (\bar{y} \equiv \bar{x}) \land (\bar{y} \Rightarrow (\bar{y} = \bar{x} \lor \bar{y} = c)) ) (1) ( (\bar{y} \equiv \bar{x}) \land (\bar{y} \Rightarrow (\bar{y} \equiv (\bar{x} \lor \bar{f})) ) (2)</td>
</tr>
<tr>
<td>( y := x \ \text{when} \ b )</td>
<td>( (\bar{y} \equiv (\bar{x} \land \bar{b} \land \bar{b})) \land (\bar{y} \Rightarrow (\bar{y} = \bar{x})) ) (1) ( (\bar{y} \equiv (\bar{x} \land \bar{b} \land \bar{b})) \land (\bar{y} \Rightarrow (\bar{y} \equiv \bar{x})) ) (2)</td>
</tr>
<tr>
<td>( y := x \ \text{default} \ z )</td>
<td>( (\bar{y} \equiv (\bar{x} \lor \bar{z})) \land (\bar{y} \Rightarrow ((\bar{x} \land (\bar{y} = \bar{x})) \lor (\sim \land \bar{y} = \bar{z}))) ) (1) ( (\bar{y} \equiv (\bar{x} \lor \bar{z})) \land (\bar{y} \Rightarrow ((\bar{x} \land (\bar{y} = \bar{x})) \lor (\sim \land (\bar{y} = \bar{z}))) ) (2)</td>
</tr>
<tr>
<td>( P_1 \parallel P_2 )</td>
<td>( \Phi_{P_1} \land \Phi_{P_2} )</td>
</tr>
<tr>
<td>( P \ \text{where} \ x )</td>
<td>( \exists x, \exists x \cdot \Phi_P )</td>
</tr>
</tbody>
</table>
The 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 abstraction 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.

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.

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.

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.

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: \( \text{level, zlevel} \in [1, +\infty], \text{faucet, zfaucet} \in [0, \infty], \text{pump, zpump} \in [0, +\infty] \).

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 \( \Phi_{\text{Bathtub}} \) is the one already provided in Section 4.2. Furthermore, we have to generate clock synchronization and clock emptiness properties (formula \( \varphi \)) from the program. Among these properties, let us focus on the following:

1. \( \text{pump} \) and \( \text{faucet} \) have disjoint clocks: \( \sim(\text{faucet} \land \text{pump}) \)
2. The water cannot overflow and be scarce at the same time: \( \sim(\text{scarce} \land \text{overflow}) \)
3. \( \text{alarm} \) and \( \text{level} \) have the same clock: \( \text{alarm} \Leftrightarrow \text{level} \)

In the third step, we encode the formula \( \Phi_{\text{Bathtub}} \land \sim \varphi \), where \( \varphi \) denotes the property to be checked. For instance, to check whether or not the signals \( \text{alarm} \) and \( \text{level} \) are synchronous, we use \( \varphi = \text{alarm} \Leftrightarrow \text{level} \). With the Yices SMT-solver, we get \( \text{unsat} \), which means that \( \Phi_{\text{Bathtub}} \models \varphi \). Thanks to Proposition 1, the property \( \varphi \) 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 \( \varphi \). For property "the water cannot overflow and be scarce at the same time", a possible concretisation is the program:

\[ \text{true when scarce when overflow} = 0. \]

Finally, we obtain the process in Fig. 7.

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 \( \text{ghost\_alarm} \) signal is always equal to \( \text{false} \), as illustrated in Fig. 8. This is represented at lines 08 and 09.

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 \( \text{ghost\_alarm} \) signal is always equal to \( \text{false} \).
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.

---

**Figure 6.** Bathtub integer kernel, after compilation.

**Figure 9.** Bathtub model composed with all clock constraints.

---

01: process Bathtub_Ter =
02: 01: if (clk_2)
03: 02: { // integer level; boolean alarm, ghost_alarm; }
04: 03: { // level := zlevel + faucet - pump
05: 04: ... | ghost_alarm := (true when scarce when overflow)
06: 05: 13: [ ] default false
07: 06: 14: [ ] true when scarce when overflow = 0
08: 07: 15: [ ] faucet = pump = 0
09: 08: 16: [ ] alarm = level
10: 09: 17: where
11: 10: 18: integer zlevel, zfaucet, zpump, faucet, pump;
12: 11: 19: boolean overflow, scarce;
13: 12: 20: end;
14: 13: 01: { ghost_alarm = FALSE; 02: 01: /* produce output value
15: 02: for the signal ghost_alarm */ } ...

---

**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.

---

**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 overload 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 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 $b_1$ and $b_2$ of event type, are defined according to the value of an integer signal $i$. The event $b_1$ occurs when the value of $i$ is between $-10$ and 10, while $b_2$ occurs when $i$ is between $-5$ and 5. One can straightforwardly deduce that the clock of $b_2$ is a subset of the clock of $b_1$.

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 $b_1$ and $b_2$ appear to be sub-clocks of that the clock of $i$, the clock hierarchy between $b_1$ and $b_2$ is not reflected. This leads to a control structure in generated code where the trigger testing related to $b_2$ is always performed, even though that of $b_1$ is false while it is unnecessary.

Actually the above situation can be avoided for an optimized code by enabling a finer analysis (not with a Boolean abstraction, but with an integer). The formula to be proven (fourth step in the proposed flow) encodes the clock inclusion of $b_1$ and $b_2$ (in SIGNAL, $b_1 \Rightarrow b_2$ is encoded as $b_1 \Rightarrow b_2$):

$$
\begin{align*}
(b_1 &\Rightarrow (i > 10 \land i < 10)) \\
(b_2 &\Rightarrow (i > 5 \land i < 5)) \\
((b_1 \land b_2) &\Rightarrow b_2)
\end{align*}
$$

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.

Figure 14. Clock hierarchy for Inclusion process.

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 CE.

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