Precise deadlock detection for polychronous data-flow specifications

van Chan Ngo, Jean-Pierre Talpin, Thierry Gautier

Precise Deadlock Detection for Polychronous Data-flow Specifications

Van Chan Ngo Jean-Pierre Talpin Thierry Gautier

INRIA Rennes-Bretagne Atlantique, 35042 Campus Beaulieu, Rennes, France
Email: *firstname.lastname@inria.fr*

***Abstract*—Dependency graphs are a commonly used data structure to encode the streams of values in data-flow programs and play a central role in scheduling instructions during automated code generation from such specifications. In this work, we propose a precise and effective method that combines a structure of dependency graph and first order logic formulas to check whether multi-clocked data-flow specifications are *deadlock free* before generating code from them. We represent the flow of values in the source programs by means of a dependency graph and attach first-order logic formulas to condition these dependencies. We use an SMT solver to effectively reason about the implied formulas and check deadlock freedom.**

***Keywords*—Deadlock detection, compilation, polychrony, dependency graphs, SMT solving, formal verification.**

I. Introduction

The synchronous languages such as Esterel [4], Lustre [13] and Signal [8] have been successfully used to design safety-critical real-time systems. Synchronous programming provides a precise semantics of programs, and the system properties can be proved mathematically by using formal methods. For instance, the SCADE tool is used by European industry to develop the control and display systems of Airbus A380. In the synchronous programming framework, the computations (in particular computations about time) are considered to have zero duration which, in practice, is acceptable as long as operations have a bounded duration. An in-depth introduction to the synchronous paradigm resulting of this hypothesis may be found in [3].

Problem statement. The analysis presented in this paper considers the Signal language as part of the instrumentation of the Eclipse project POP[1] within a translation validation tool chain [19]. The Signal compiler constructs the data dependencies among signals in a program, represented as a labeled directed graph, in which the labels are polynomials in $\mathbb{Z}/3\mathbb{Z}$. Then for each dependency cycle in the graph, it checks the product of the cycle labels. If this product, which is represented as a polynomial, is equal to the *null clock*, then it does not stand for a deadlock. However, consider the *sampling* operator $y := x \ when \ b$, in which the clock of the signal y is defined by the *condition clock* $[b]$, which defines the instants where b is present and has value *true*. If the Boolean expression b is a non Boolean relation (e.g. a comparison between numerical expressions), then the $\mathbb{Z}/3\mathbb{Z}$ abstraction considers the expression's clock instead. This yields an approximation of the actual dependency which may cause the compiler to approximate dependencies, like in the example above.

Contribution. We propose a more precise deadlock detection approach for deadlock-free checking of synchronous programs defining multi-clocked embedded real-time systems in the Signal language. Our approach permits the compiler to detect deadlocks more precisely when dealing with numerical expressions. In our solution, the data dependencies among signals are represented by *Synchronous Data-flow Dependency Graphs* (SDDGs). A SDDG is a labeled directed graph in which each node is a signal or clock variable and each edge represents a dependency between two nodes. Each edge is labelled by a condition at which the dependency is effective. We borrow the Boolean-interval abstraction from [10] to encode the clock labels. That means every signal is associated with a pair of the form *(clock, value)*, where *clock* is a Boolean function and *value* is a Boolean or numerical function, abstracted as an interval. We use a SMT solver to reason on the labels when deciding a dependency cycle in a SDDG to stand for a deadlock. We show how our approach addresses the limitation of the current deadlock detection used in the Signal compiler through a concrete example.

Outline. The remainder of this paper is organized as follows. Section II introduces the Signal language. Section III provides an overview of the current deadlock detection approach used in the Signal compiler and shows its limitation when dealing with numerical expressions. In Section IV, we present the concept of SDDG as well as the Boolean-interval abstraction and deadlock detection in SDDGs. This section also shows how we use a SMT solver to implement the detecting deadlock procedure. Section V presents related works and concludes.

II. The Signal Language

Signal is a polychronous data-flow language that allows the specification of multi-clocked systems, called *polychrony model*. Signal handles unbounded sequences of typed values $x(t)_{t\in\mathbb{N}}$, called *signals*, denoted as x. Each signal is implicitly indexed by a logical *clock* indicating the set of instants at which the signal is present, noted C_x. At a given instant, a signal may be present where it holds a value, or absent where it holds no value (denoted by $\#$). Given two signals, they are *synchronous* if and only if they have the same clock. In Signal, a process (written P or Q) consists of the synchronous composition (noted $|$) of equations over signals x, y, z, written $x := y \ op \ z$ or $x := op(y, z)$, where op is an operator. Then a program is a process and the language is modular. In particular, a process can be used as a basic pattern, by means

[1]Polychrony on Polarsys: an Eclipse project of the Polarsys Industry Working Group, http://www.polarsys.org/projects/polarsys.pop

of an interface that describes its parameters and its input and output signals. Moreover, a process can use other subprocesses, or even external parameter processes that are only known by their interfaces.

Data Domains. Data types consist of usual scalar types (Boolean, integer, float, complex, and character), enumerated types, array types, tuple types, and the special type *event*, subtype of the Boolean type which has only one value, *true*.

Operators. The *core language* consists of two kinds of "statements" defined by the following primitive operators: four operators on signals and two operators on processes. The operators on signals define basic processes (with implicit clock relations) while the operators on processes are used to construct complex processes with the parallel composition operator:

- *Stepwise Functions:* $y := f(x_1, ..., x_n)$, where f is a *n-ary* function on values, defines the extended stream function over synchronous signals as a basic process whose output y is synchronous with $x_1, ..., x_n$ and $\forall t \in C_y, y(t) = f(x_1(t), ..., x_n(t))$.
- *Delay:* $y := x\$1\ init\ a$ defines a basic process such that y and x are synchronous, $y(0) = a$, and $\forall t \in C_y \wedge t > 0, y(t) = x(t-1)$.
- *Merge:* $y := x\ default\ z$ defines a basic process which specifies that y is present if and only if x or z is present, and that $y(t) = x(t)$ if $t \in C_x$ and $y(t) = z(t)$ if $t \in C_z \setminus C_x$.
- *Sampling:* $y := x\ when\ b$ where b is a Boolean signal, defines a basic process such that $\forall t \in C_x \cap C_b \wedge b(t) = true, y(t) = x(t)$, and otherwise, y is absent.
- *Composition:* If P_1 and P_2 are processes, then $P_1 \mid P_2$, also denoted as $(|P_1 \mid P_2|)$, is the process resulting of their parallel composition. This process consists of the composition of the systems of equations. The composition operator is commutative, associative, and idempotent.
- *Restriction:* $P\ where\ x$, where P is a process and x is a signal, specifies a process by considering x as local variable to P (i.e., x is not accessible from outside P).

Clock Relations. Clock relations can be defined explicitly: $y := \hat{}x$ specifies that y is the clock of x. The synchronization $x \hat{}= y$ means that x and y have the same clock. The clock extraction from a Boolean signal is denoted by a unary *when*: $when\ b$. The clock union $x \hat{}+ y$ defines a clock as the union $C_x \cup C_y$. In the same way, the clock intersection $x \hat{}* y$ and the clock difference $x \hat{}- y$ define clocks $C_x \cap C_y$ and $C_x \setminus C_y$.

Example. The following Signal program illustrates the language syntax and meaning of primitive operators. It emits an integer sequence v whose ith value v_i is the double of the input x_i depending on the other input c.

```
1 process CycleDependency=
2 (? integer x, c; ! integer v)
3  (| y := (v when (c <= 0)) default x
4   | u := y + x
5   | v := u when (c >= 1)
6   |)
7 where integer y, u end;
```

In this program, x, c and v at lines (2) and (3) are respectively input and output signals of type *integer*. Line (4) expresses that y holds the value of v if v, c are present and c is less than or equal to 0, otherwise it holds the value of x when x is present. At line (5), u, y, x are constrained to be synchronous. And u is set to the sum of y and x when y and x are present. Line (6) defines that v is set to u when c is greater than or equal to 1. Line (8) indicates that y and u are local signals. A possible run of the program is depicted by the following trace:

```
1 x   1  3  2  2  #  #  4  7  9 ...
2 c   1  3  0  1 -1 -2  3  6  2 ...
3 y   1  3  2  2  #  #  4  7  9 ...
4 u   2  6  4  4  #  #  8 14 18 ...
5 v   2  6  #  4  #  #  8 14 18 ...
```

III. Deadlock Detection in the Signal Compiler

Before generating the executable code on a given architecture, the compilation performed by the Signal compiler aims at proving the *reactivity* and the *determinism* of programs by modeling the synchronization relations and checking the absence of cyclic data dependencies in program specifications. In our consideration, the compiler needs to answer the following question: *"Is the program deadlock-free"*. To answer this question, the Signal compiler uses the *Graph of Conditional Dependencies* (GCD) in which a dependency between data is conditioned by a clock. We will illustrate this technique with an example (a more detailed discussion is presented in [17]).

The Synchronization Space. The state of a Boolean signal x can be encoded in the finite field $\mathbb{Z}/_3\mathbb{Z}$ of integers modulo 3 as the following values: -1: *present* and *false*, 1: *present* and *true*, and 0: *absent*. Then, the clock of the signal x may be clearly represented by x^2. For non-Boolean signals: their presence is encoded as 1 and their absence as 0. This principle is used to represent synchronization relations in a Signal program. The coding of the primitive operators is deduced from their definition as described in [9]. For instance, applying this principle on program `CycleDependency`, we have the following algebraic coding, where we introduce three fresh variables c_1, c_2 and v_1 to replace the expressions $(c <= 0), (c >= 1)$ and $(v\ when\ c_1)$:

$$y^2 = v_1^2 + (1 - v_1^2)x^2;\ v_1^2 = v^2(-c_1 - c_1^2)$$
$$u^2 = y^2 = x^2; c_1^2 = c^2 = c_2^2;\ v^2 = u^2(-c_2 - c_2^2)$$

Graph of Conditional Dependencies. The GCD calculated by the Signal compiler is a labeled directed graph where the vertices are the signals and clock variables, the edges indicate data dependencies among signals and clock variables, the labels are polynomials on $\mathbb{Z}/_3\mathbb{Z}$ which represent the conditions at which the dependencies are valid. In case of the program `CycleDependency`, the dependencies among signals y, u and v are depicted as the sub-graph in Figure 1. The notation $x \rightarrow^{c^2} y$ means that at a given instant, y depends on x iff $c^2 = 1$.

Limitations. Based on GCD graphs, the Signal compiler identifies the potential deadlocks in the program. Such a bad dependency between signals will appear as a cycle in the graph if the product of the labels of its edges is not the null clock. Then, we say that there exists a deadlock in the program. The deadlock detection can be performed on a GCD

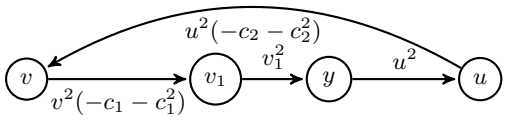

Fig. 1: Dependencies among y, u, v

as follows. Given a GCD, we say that the graph is deadlock-free iff for every cycle $(x_1, ..., x_n, x_1)$ in the graph, the product of $c_1, ..., c_n$ is the null clock (equal to 0), where c_i is the label of the edge between x_i and x_{i+1}, and c_n is the label of the edge between x_n and x_1. However, for the sampling operator $y := x\ when\ b$, if the Boolean expression b is a non Boolean relation, then the $\mathbb{Z}/_3\mathbb{Z}$ encoding considers this condition clock as an indeterminate value. This naturally yields over-approximated detection when dealing with non Booleans values.

The `CycleDependency` program and its dependencies among the signals y, u and v in Figure 1 exemplifies the over-approximated detection (we omit the dependencies among v, v_1, y and u). These dependencies introduce a cycle in the graph. To verify that this cycle is not a deadlock, the Signal compiler calculates the product of the labels as $x^2(-c_2 - c_2^2)(-c_1 - c_1^2) * x^2(-c_2 - c_2^2)(-c_1 - c_1^2) * x^2 * x^2(-c_2 - c_2^2)$. With the current clock calculus, the compiler concludes that this cycle may cause a deadlock since at some instants the above product is different from 0 (e.g. when the signals x, c_1, c_2 are present and both c_1 and c_2 hold the value *true*, or $x^2 = 1, c_1 = 1$, and $c_2 = 1$). But in fact, c_1 and c_2 cannot hold the value *true* at the same time.

The above limitation makes the Signal compiler reject some possibly valid programs. To address this issue, we propose a new deadlock detection to better handle numerical operators.

IV. A More Precise Deadlock Detection

In this section, we will present a more precise deadlock detection technique compared to the technique currently used by the Signal compiler. This technique uses the concept of *Synchronous Data-flow Dependency Graph* (SDDG) to express the data dependencies among the signals in a program. A SDDG represents a synchronous program as a labelled directed graph in which each node is a signal or a clock and each edge from a node to another node represents the dependency between nodes. Each edge is labeled by a clock constraint. The abstraction which is used to encode the clock constraints was originally proposed by Gamatié et al. [10].

A. Signal Program as Synchronous Data-flow Dependency Graph

1) A Boolean-interval Abstraction for Clock Semantics: Let $X = \{x_1, ..., x_n\}$ be the set of all signals in program P. With each signal x_i, we attach a Boolean variable $\widehat{x_i}$ to encode its clock and a variable $\widetilde{x_i}$ of same type as x_i to encode its value. Formally, the abstract values which represent the semantics of the program can be computed using the following functions:

$\hat{}: X \longrightarrow \mathbb{B}$ associates a signal with a Boolean value
$\tilde{}: X \longrightarrow \mathbb{D}$ associates a signal with a value of the same type

The composition of Signal processes corresponds to logical conjunctions. Thus the abstract model of P will be a conjunction $\Phi(P) = \bigwedge_{i=1}^{n} \phi(eq_i)$ whose atoms are $\widehat{x_i}, \widetilde{x_i}$, where $\phi(eq_i)$ is the abstraction of statement eq_i (statement using the Signal primitive operators), and n is the number of statements in the program. In the following, we present the abstraction corresponding to each Signal operator. There are two definitions of Φ according to the type of the signal on the left hand side in each equation: (1) stands for numerical type and (2) is for logical type. The Boolean-interval abstraction preserves the behaviors of the program being abstracted, in other word, it is sound. The details of the abstraction and the soundness proof are presented in [10].

Stepwise Functions. The functions which apply on signal values in the primitive *stepwise functions* are usual logic operators (*not, and, or*), numerical comparison functions ($<, >, =, <=, >=, /=$), and numerical operators ($+, -, *, /$) (denoted by $\square$). In our implementation, we replace the operation results by intervals and their representation in logic context by *uninterpreted functions*.

The abstraction $\phi(y := f(x_1, ..., x_n))$ of stepwise functions is defined as follows:

$$\bigwedge_{i=1}^{n}(\hat{y} \Leftrightarrow \hat{x_i}) \wedge (\hat{y} \Rightarrow \tilde{y} \in \phi(f(x_1, ..., x_n))) \quad (1)$$
$$\bigwedge_{i=1}^{n}(\hat{y} \Leftrightarrow \hat{x_i}) \wedge (\hat{y} \Rightarrow \tilde{y} \Leftrightarrow \phi(f(x_1, ..., x_n))) \quad (2)$$

where the abstraction $\phi(f(x_1, ..., x_n)$ is defined by induction as follows: $\phi(true) = true$ and $\phi(false) = false$; $\phi(x) = \tilde{x}$ if x is of Boolean type, $\phi(x) = true$ if x is of event type; $\phi(x) = inv(x)$ if x is of non-Boolean type, where $inv(x)$ is the interval of x; $\phi(x_1\ and\ x_2) = \widetilde{x_1} \wedge \widetilde{x_2}$; $\phi(x_1\ or\ x_2) = \widetilde{x_1} \vee \widetilde{x_2}$; $\phi(not\ x_1) = \neg\widetilde{x_1}$; $\phi(x <= c) = \tilde{x} \in (-\infty, c]$, $\phi(x < c) = \tilde{x} \in (-\infty, c)$; $\phi(x_1 <= x_2) = (\tilde{x} \in \phi(x_1 - x_2)) \wedge \tilde{x} \in (-\infty, 0]$, where x is a fresh variable; $\phi(x_1 \square x_2) = \widetilde{\square}(\phi(x_1), \phi(x_2))$, an approximation of numerical operations on intervals, corresponding to $\square$ as in [2]. For example, for addition operation $i \tilde{+} j \equiv [i^- + j^-, i^+ + j^+]$, where i^- and i^+ are respectively the lower and upper bounds of the interval i.

Delay. The encoding of the *delay* is given as follows. This encoding requires that at any instant, signals x and y have the same clock. If they are numerical signals, then they have the same interval. Otherwise, we introduce a memorization variable $m.x$ that stores the last value of x. The next value of $m.x$ is $m.x'$ and it is initialized to a in $m.x_0$.

$$(\hat{y} \Leftrightarrow \hat{x}) \wedge (\hat{y} \Rightarrow (\tilde{y} \in \phi(x) \vee \tilde{y} = a)) \quad (1)$$
$$(\hat{y} \Leftrightarrow \hat{x}) \wedge (\hat{y} \Rightarrow (\tilde{y} \Leftrightarrow \widetilde{m.x} \wedge \widetilde{m.x'} \Leftrightarrow \tilde{x})) \wedge (\widetilde{m.x_0} \Leftrightarrow a) \quad (2)$$

Merge and sample. The encoding of the *merge* and *sample* contributes to $\Phi(P)$ with the following propositions, respectively:

$$(\hat{y} \Leftrightarrow \hat{x} \vee \hat{z}) \wedge (\hat{y} \Rightarrow (\hat{x} \wedge (\tilde{y} = \tilde{x})) \vee (\neg\hat{x} \wedge (\tilde{y} = \tilde{z}))) \quad (1)$$
$$(\hat{y} \Leftrightarrow \hat{x} \vee \hat{z}) \wedge (\hat{y} \Rightarrow (\hat{x} \wedge (\tilde{y} \Leftrightarrow \tilde{x})) \vee (\neg\hat{x} \wedge (\tilde{y} \Leftrightarrow \tilde{z}))) \quad (2)$$

and

$$(\hat{y} \Leftrightarrow (\hat{x} \wedge \hat{b} \wedge \tilde{b})) \wedge (\hat{y} \Rightarrow (\tilde{y} = \tilde{x})) \quad (1)$$
$$(\hat{y} \Leftrightarrow (\hat{x} \wedge \hat{b} \wedge \tilde{b})) \wedge (\hat{y} \Rightarrow (\tilde{y} \Leftrightarrow \tilde{x})) \quad (2)$$

Composition $\phi(P_1|P_2)$ is defined by $\phi(P_1) \wedge \phi(P_2)$.

Clock Relations. Given the above rules, we can obtain the following abstraction for derived operators on clocks. Here, z is a signal of type *event*. $\phi(z := \hat{x}) = (\hat{z} \Leftrightarrow \hat{x}) \wedge (\hat{z} \Rightarrow \tilde{z})$;

$\phi(x \hat{} = \ y) = \hat{x} \Leftrightarrow \hat{y}$; $\phi(z := x \hat{} + \ y) = (\hat{z} \Leftrightarrow (\hat{x} \vee \hat{y})) \wedge (\hat{z} \Rightarrow \tilde{z})$; $\phi(z := x \hat{} * \ y) = (\hat{z} \Leftrightarrow (\hat{x} \wedge \hat{y})) \wedge (\hat{z} \Rightarrow \tilde{z})$; $\phi(z := x \hat{} - \ y) = (\hat{z} \Leftrightarrow (\hat{x} \wedge \neg\hat{y})) \wedge (\hat{z} \Rightarrow \tilde{z})$; $\phi(z := \text{ when } b) = (\hat{z} \Leftrightarrow (\hat{b} \wedge \tilde{b})) \wedge (\hat{z} \Rightarrow \tilde{z})$.

Example. Assume that the variant intervals for program signals are given as $x \in (-\infty, +\infty)$, $c \in (-\infty, +\infty)$, $y \in (-\infty, +\infty)$, $u \in (-\infty, +\infty)$, $v \in (-\infty, +\infty)$. Applying the abstraction rules above, the abstraction of the `CycleDependency` program is represented by the following first-order logic formula:

$$(\hat{y} \Leftrightarrow \widehat{v_1} \vee \hat{x}) \wedge (\tilde{y} \in (-\infty, +\infty)) \wedge (\widehat{v_1} \Leftrightarrow \hat{v} \wedge \widehat{c_1} \wedge \widetilde{c_1}) \wedge (\widetilde{v_1} \in (-\infty, +\infty)) \wedge (\widehat{c_1} \Leftrightarrow \hat{c}) \wedge (\widetilde{c_1} \Leftrightarrow (\tilde{c} \in (-\infty, 0])) \wedge (\hat{u} \Leftrightarrow \hat{y} \Leftrightarrow \hat{x}) \wedge (\tilde{u} \in (-\infty, +\infty)) \wedge (\hat{v} \Leftrightarrow \hat{u} \wedge \widehat{c_2} \wedge \widetilde{c_2}) \wedge (\tilde{v} \in (-\infty, +\infty)) \wedge (\widehat{c_2} \Leftrightarrow \hat{c}) \wedge (\widetilde{c_2} \Leftrightarrow (\tilde{c} \in [1, +\infty)))$$

2) Synchronous Data-flow Dependency Graph: Consider the basic process $y := x \ default \ z$ (and the clock relations among the signals), the "valid" states are: x is present and y is present; or x is absent, z is present, and y is present; or x, y and z are absent. They can be represented by $\hat{y} \Leftrightarrow (\hat{x} \vee \hat{z})$ in our Boolean-interval abstraction. According to the valid states of the signals, the different data dependencies among signals in the basic process $y := x \ default \ z$ are depicted in Figure 2, where the labels represent the conditions at which the dependencies are effective. For instance, when $\hat{x} = true$, y is defined by x; otherwise it is defined by z when $\hat{x} = \texttt{false}$ and $\hat{z} = true$. We can see that the graph in this figure has the following property: an edge cannot exist if one of its extremity nodes is not present (or the corresponding signal holds no value). In our example, this property can be expressed in the abstraction as: $\hat{x} \Rightarrow \hat{y} \wedge \hat{x}; \neg\hat{x} \wedge \hat{z} \Rightarrow \hat{y} \wedge \hat{z}$.

A SDDG for a given program is a labelled directed graph

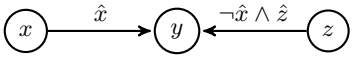

Fig. 2: The SDDG of Merge Operator

in which each node is a signal or clock variable and each edge represents the dependency between nodes. Each edge is labelled by a first-order logic formula which represents the clock at which the dependency between the extremity nodes is effective. Formally, a SDDG is defined as follows:

Definition 1: A SDDG associated with a process P is a tuple $G = \langle N, E, I, O, C, m_N, m_E \rangle$ where:

- N is a finite set of nodes, each of which represents the equation defining a signal or a clock;
- $E \subseteq N \times N$ is the set of dependencies between nodes;
- $I \subseteq N$ is the set of input nodes;
- $O \subseteq N$ is the set of output nodes;
- C is the set of first-order logic formulas over a set of clocks in the Boolean-interval abstraction;
- $m_N : N \longrightarrow C$ is a mapping labeling each node with a clock; it defines the existence condition of a node;
- $m_E : E \longrightarrow C$ is a mapping labeling each edge with a clock constraint; it defines the existence condition of an edge.

Operator	Dependencies
x	$\hat{x} \xrightarrow{\hat{x}} x, m_N(\hat{x}) = \hat{x}, m_N(x) = \hat{x}$
c (Boolean signal)	$c \xrightarrow{[c]} [c], m_N(c) = \hat{c}, m_N([c]) = [c]$ $c \xrightarrow{[\neg c]} [\neg c], m_N(c) = \hat{c}, m_N([\neg c]) = [\neg c]$
$x \xrightarrow{c} y$	$[c] \xrightarrow{[c]} y, m_N([c]) = [c], m_N(y) = \hat{y}$
$y := f(x_1, ..., x_n)$	$x_1 \xrightarrow{\hat{y}} y \ ... \ x_n \xrightarrow{\hat{y}} y$ $m_N(x_i) = \hat{x_i}, m_N(y) = \hat{y}, i = 1, ..., n$
$y := x\$1$ init a	$m_N(x) = \hat{x}, m_N(y) = \hat{y}$
$y := x$ when b	$x \xrightarrow{\hat{y}} y, m_N(x) = \hat{x}, m_N(y) = \hat{y}$ $b \xrightarrow{\hat{y}} \hat{y}, m_N(b) = \hat{b}, m_N(\hat{y}) = \hat{y}$
$y := x$ default z	$x \xrightarrow{\hat{x}} y, m_N(x) = \hat{x}, m_N(y) = \hat{y}$ $z \xrightarrow{\hat{z} \wedge \neg\hat{x}} y, m_N(z) = \hat{z}, m_N(y) = \hat{y}$

TABLE I: The Dependencies of the Core Language

The clock labeling in SDDG provides a dynamic dependency feature. This clock imposes the following property: *An edge exists if its two extremity nodes exist*. This property can be translated in our Boolean-interval abstraction as: $\forall (x, y) \in E, m_E(x, y) \Rightarrow (m_N(x) \wedge m_N(y))$. We denote the fact that there exists a dependency between two nodes (signals or clocks) x and y at a clock condition $m_E(x, y) = \hat{c}$ by $x \xrightarrow{\hat{c}} y$. A *dependency path* from x to y is any set of nodes $s = \{x_0, x_1, ..., x_k\}$ such that $x = x_0 \xrightarrow{\widehat{c_0}} x_1 \xrightarrow{\widehat{c_1}} ... \xrightarrow{\widehat{c_{k-1}}} x_k = y$.

In Table I, we construct the dependencies among signals for the core language, where the sub-clocks $[c]$ and $[\neg c]$ are encoded as $\hat{c} \wedge \tilde{c}$ and $\hat{c} \wedge \neg\tilde{c}$, respectively, in our abstraction. The edges are labelled by clocks which are represented by a first-order logic formula in our abstraction. The dependencies in this table impose the implicit properties for a SDDG, for instance, the basic process $y := x \ when \ b$ corresponding to the primitive operator Boolean sampling satisfies $\hat{y} \Rightarrow \hat{x} \wedge \hat{y}$ and $\hat{y} \Rightarrow \hat{b} \wedge \hat{y}$.

We also assume that all considered programs are written with the primitive operators, meaning that derived operators are replaced by their definition with primitive ones, and there are no nested operators (these nested operators can be broken by using fresh signals).

Following the above construction rules, we can obtain the SDDG in Figure 3, for the simple program `CycleDependency` (we omit the parts of graph that represent the dependencies of c_1 and c_2 on c). In this graph, the clock labels are defined in the above Boolean-interval abstraction.

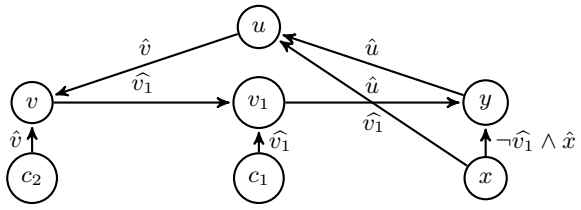

Fig. 3: The SDDG of CycleDependency

B. Precise Deadlock Detection

In this section, we will present a more precise deadlock detection technique which is based on the concept of SDDG

along with the Boolean-interval abstraction. We will show a performance comparison of our technique with the current technique of the Signal compiler using a concrete example.

1) Deadlock Definition: Let G be a SDDG, and $x \xrightarrow{\hat{c}} y$ be an edge in G. Assume that there exists a path from y to x, that forms a dependency cycle between x and y. A cycle of dependencies, standing for a deadlock, is defined with a similar meaning to the one in GCD. We say that such cycle is a deadlock in G iff the dependencies of x to y and vice-versa are effective at the same time. Transposing the notion of deadlock to SDDG graphs, we have the following definition:

Definition 2: Let $G = \langle N, E, I, O, C, m_N, m_E \rangle$ be a SDDG; a cycle $p_c = x_1, ..., x_n, x_1$ in G is said a deadlock if there exists an interpretation such that the formula $m_E(x_1, x_2) \wedge m_E(x_2, x_3) \wedge ... \wedge m_E(x_n, x_1)$ is *true*.

Based on the definition of deadlock in SDDG graphs, we can have a consequent definition of *deadlock-free* of a SDDG as follows:

Definition 3: An SDDG $G = \langle N, E, I, O, C, m_N, m_E \rangle$ is deadlock-free iff every cycle $(x_1, ..., x_n, x_1)$ in G satisfies:

$$m_E(x_1, x_2) \wedge m_E(x_2, x_3) \wedge ... \wedge m_E(x_n, x_1) \Leftrightarrow false$$

Obviously, the fact that the conjunction of first-order logic formulas, in which the dependencies are effective when they are valid, is *false* indicates that a deadlock does not exist if all the dependencies of a cycle in the SDDG graph cannot be present at the same time.

2) Proving Deadlock Freedom by SMT Solver: Given the SDDG of a program P, we introduce an approach to check whether the graph is deadlock-free. It is implemented with a SMT solver [7], [20]. A SMT solver decides the satisfiability of arbitrary logic formulas of linear real and integer arithmetic, scalar types, other user-defined data structures, and uninterpreted functions. If the formula belongs to the decidable theory, the solver gives two types of answers: `sat` when the formula has a model (there exists an interpretation that satisfies it); or `unsat` otherwise. In our case, the formulas which label the edges of the graph are over Boolean variables and uninterpreted functions, thus the solving is decidable and very efficient [1], [5].

Following Definition 3, we will traverse the entire graph to find all cycles and verify that every cycle does not stand for a deadlock. Notice that here, we do not provide any specific algorithm to find the cycles in a directed graph, interested readers can refer to any research on this problem (e.g. the work of D.B. Johnson [14]). It means that the basic element we have to prove is that given a dependency cycle and the conjunction of its labels, this conjunction formula is always evaluated to the value *false*.

Consider a dependency cycle $x_1 \rightarrow^{\widehat{c_1}} x_2 \rightarrow^{\widehat{c_2}} ... \rightarrow^{\widehat{c_{n-1}}} x_n \rightarrow^{\widehat{c_n}} x_1$. This cycle does not stand for a deadlock iff the formula $\bigwedge_{i=0}^{n} \widehat{c_i} \Leftrightarrow false$ is valid within the logical context defined by the abstraction of P. The checking of this condition can be implemented by asking a SMT solver to check $M \not\models \neg(\bigwedge_{i=0}^{n} \widehat{c_i} \Leftrightarrow false)$. Based on our experience with SMT solvers, it is more efficient to make the conjunction of all formulas such as $\bigwedge_{i=0}^{n} \widehat{c_i} \Leftrightarrow false$ and call the solver once.

3) Implementation and Illustrative Example: We describe the main steps of our approach, and the techniques we use to implement them. The tool's flow is depicted in Figure 4. First, it takes the input program P, and constructs the corresponding SDDG graph. It finds all cycles in the graph. Finally, in the solving phase, it checks the validity of the formula $(\bigwedge_{i=0}^{n} \widehat{c_i} \Leftrightarrow false)$ for each cycle $(x_1, ..., x_n, x_1)$.

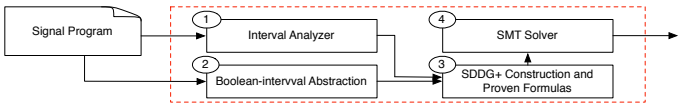

Fig. 4: Block diagram of the analytic flow

Interval Analyzer. This step determines the interval of every signal in the program. For every input signal, it is assumed that its interval is known. This step applies the algorithm presented in [11]. The tool in [16] is used to compute an over-approximation of the variation interval of each numerical signal.

Abstraction. The input program is encoded according to the Boolean-interval abstraction scheme in Section IV-A1. The output of this step is a first-order logic formula.

SDDG Construction. After obtaining the interval analysis and the abstract model of the input program, this step will construct the corresponding SDDG graph according to the rules in Table I. The labels of edges in the graph are encoded as first-order logic formulas based on the above Boolean-interval abstraction scheme. In this step, the tool also detects all dependency cycles in the graph and produces the conjunction formula of all clock labels for each cycle as input for the solver.

SMT-based Proof. We delegate checking the validity of the formulas to a SMT solver. Our implementation uses the SMT-LIB common format [6] to encode the formulas as input to the SMT solver. For our experiments, we consider the Yices [7] solver, which is one of the best solvers at the SMT-COMP competition [20].

Let us illustrate the above deadlock detection technique on the `CycleDependency` program. In the first step, we will determine the variation interval for all signals in the program. We assume that the variation intervals for input signals x and c are $(-\infty, +\infty), (-\infty, +\infty)$. After the analysis, we get: $x \in (-\infty, +\infty)$, $c \in (-\infty, +\infty)$, $y \in (-\infty, +\infty)$, $u \in (-\infty, +\infty)$, $v \in (-\infty, +\infty)$. In the second step, Boolean-interval abstraction, our tool will translate the program into a logic formula Φ according to the above abstract scheme. Let us focus on the clocks of signals y, u, and v, which are given as follows:

$$\hat{y} \Leftrightarrow \widehat{v_1} \vee \hat{x}; \hat{u} \Leftrightarrow \hat{y} \Leftrightarrow \hat{x}; \widehat{c_1} \Leftrightarrow \widehat{c_2} \Leftrightarrow \hat{c}$$
$$\widehat{v_1} \Leftrightarrow \hat{v} \wedge \widehat{c_1} \wedge (\tilde{c} \in (-\infty, 0]); \hat{v} \Leftrightarrow \hat{u} \wedge \widehat{c_2} \wedge (\tilde{c} \in [1, +\infty))$$

In the third step, we construct the SDDG graph in Figure 3. Here, we detect that there exists a dependency cycle (v, v_1, y, u, v) in the graph, the tool then generates the formula $(\widehat{v_1} \wedge \widehat{v_1} \wedge \hat{u} \wedge \hat{v}) \Leftrightarrow false$, referred to as φ, to delegate to the SMT solver. In the logical context defined by Φ, replacing the definitions of $\widehat{c_1}, \widehat{c_2}, \hat{u}, \hat{v}$ and $\widehat{v_1}$ in the program abstraction

model, we get:

$$(\hat{x} \wedge \hat{c} \wedge (\tilde{c} \in [1, +\infty)) \wedge \hat{c} \wedge (\tilde{c} \in (-\infty, 0]) \wedge \hat{x} \wedge \hat{c} \wedge (\tilde{c} \in [1, +\infty)) \wedge \hat{c} \wedge (\tilde{c} \in (-\infty, 0]) \wedge \hat{x} \wedge \hat{x} \wedge \hat{c} \wedge (\tilde{c} \in [1, +\infty))) \Leftrightarrow false$$

With the Yices solver, we will get `unsat` when checking the satisfiability of $\neg\varphi$, which means that φ is valid. Thus, the graph is deadlock-free.

Our deadlock detection technique is more precise than the current technique used by the Signal compiler when dealing with numerical expressions. It admits less erroneous (or "spurious") decision on deadlock detection than the current technique. That means that when our approach is applied on the Signal compiler, it will make the compiler avoid rejecting valid programs. The reason why our technique is more expressive than the current one is that it uses a more suitable and precise abstraction for numerical expressions. For instance, here, our tool can detect that the two signals v and v_1 cannot be present at the same time.

V. Conclusion and Related Work

The concept of interval-Boolean abstraction was introduced in [10] by A. Gamatié et al. to provide a more efficient static analysis of Signal compiler. In that work, the authors aim to make the analysis of clock hierarchy more powerful when dealing with the numerical expressions. The analysis presented in this paper is implemented as part of the instrumentation of the Eclipse project POP on synthesizing safety wrappers for polychronous specifications through polyhedral analysis, and generalizes the work presented in [15] on false-loop detection using SMT-solving to the case of a polyhedral analysis.

A relevant study on synchronous data-flow dependency graph [19] concerns the preservation of data dependency in the transformation of Signal programs during the compilation process. In that work, the authors represent the data dependencies in the source synchronous program and the compiled program as labelled directed graphs, in which the edges are labeled by first-order logic formulas. Given two graphs, a *correct transformation* relation between them is defined, which expresses the semantic preservation of data dependency. In implementation, a SMT-solver is used for checking the existence of the correct transformation relations. This validation is applied to the second compilation phase of the Signal compiler, *static scheduling*.

In this paper, we propose a more precise deadlock detection approach for deadlock-free checking of synchronous programs written in Signal language. Our approach permits the compiler avoiding emitting spurious decision on deadlock detection while the current technique does when dealing with numerical expressions. In our solution, the data dependencies among signals are represented by SDDG graphs, in which the nodes are signals or clock variables, edges are dependencies among signals. Each edge is labeled by a condition expressed as a first-order logic formula at which the dependency is effective. We use a SMT solver to reason on the labels when deciding whether a dependency cycle in a SDDG stands for a deadlock. We have provided a technique to implement and integrate our deadlock detection approach within the Polychrony toolset with Yices. As next work, we will do some enhancement of the implementation by making it a fully automated process. We will take more case studies as well.

References

[1] W. Ackerman. Solvable cases of the Decision Problem. *Study in Logic and the Foundations of Mathematics*. North-Holland, Amsterdam, 1954.

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

[3] A. Benveniste, and G. Berry. The Synchronous Approach to Reactive and Real-time Systems. *In Processing of the IEEE, Vol 79(9)*. September 1991.

[4] G. Berry. The Foundations of Esterel. *In Proof, Language and Interaction: Essay in Honor of Robin Milner, MIT Press*. 2000.

[5] 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*. ISBN 978-1-5860-3929-5, 2009.

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

[7] B. Dutertre, and L. de Moura. Yices sat-solver. *http://yices.csl.ri.com*. 2009.

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

[9] A. Gamatié, T. Gautier, P. Le Guernic, and J-P. Talpin. Polychronous Design of Embedded Real-time Applications. *In ACM Transactions on Software Engineering and Methodology*. November 2005.

[10] P. Feautrier, A. Gamatié, and L. Gonnord. Enhancing the Compilation of Synchronous Data-flow Programs with Combined Numerical-Boolean Abstraction. *In CSI Journal of Computing, 1(4):86-99*. 2012.

[11] L. Gonnord, and N. Halbwachs. Abstract Acceleration to Improve Precision on Linear Relation Analysis. *Research Report, Verimag*. March, 2010.

[12] 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.

[13] 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)*. Jul 2005.

[14] D.B Johnson. Finding All the Elementary Circuits of A Directed Graph. *In SIAM J. Comput*. Vol. 4 No. 1. March 1975.

[15] B. Jose, A. Gamatié, J. Ouy, and S. Shukla. SMT based false causal loop detection during code synthesis from polychronous specifications. 9th ACM-IEEE International Conference on Formal Methods and Models for Codesign. IEEE, 2011.

[16] G. Lalire, M. Argoud, and B. Jeannet. Interproc: An Interprocedural Analyzer for Imperative Languages. *http://pop- art.inrialpes.fr/people/bjeannet/bjeannet- forge/interproc*. 2009.

[17] O. Maffeïs, and P. Le Guernic. Distributed Implementation of Signal: Scheduling and Graph clustering. *Formal Techniques in Real-Time and Fault-Tolerant Systems*. LNCS 863, Springer Verlag, 1994.

[18] M. Nanjundappa, M. Kracht, J. Ouy, and S. Shukla. Synthesizing Embedded Software with Safety Wrappers through Polyhedral Analysis in a Polychronous Framework. *Electronic System Level Synthesis Conference*. IEEE, 2012.

[19] V.C. Ngo, J-P. Talpin, T. Gautier, P. Le Guernic, and L. Besnard. Formal Verification of Synchronous Data-flow Program Transformations Toward Certified Compilers. *In Frontiers of Computer Science. Special Issue on Synchronous Programming*. Springer Verlag, 2013.

[20] A. Stump, and M. Deters. SMT-Comp. *http://www.smtcomp.org*. 2009.