Front. Comput. Sci., 2016, 10(1): 37–53
DOI 10.1007/s11704-015-4364-y

RESEARCH ARTICLE

Towards a verified compiler prototype for the synchronous language SIGNAL

Zhibin YANG[1,2,3], Jean-Paul BODEVEIX[2], Mamoun FILALI[2], Kai HU (✉)[3], Yongwang ZHAO[3], Dianfu MA[3]

1 College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, Nanjing 210016, China
2 IRIT-CNRS, Université de Toulouse, Toulouse 31062, France
3 State Key Laboratory of Software Development Environment, Beihang University, Beijing 100191, China

Abstract SIGNAL belongs to the synchronous languages family which are widely used in the design of safety-critical real-time systems such as avionics, space systems, and nuclear power plants. This paper reports a compiler prototype for SIGNAL. Compared with the existing SIGNAL compiler, we propose a new intermediate representation (named S-CGA, a variant of clocked guarded actions), to integrate more synchronous programs into our compiler prototype in the future. The front-end of the compiler, i.e., the translation from SIGNAL to S-CGA, is presented. As well, the proof of semantics preservation is mechanized in the theorem prover Coq. Moreover, we present the back-end of the compiler, including sequential code generation and multi-threaded code generation with time-predictable properties. With the rising importance of multi-core processors in safety-critical embedded systems or cyber-physical systems (CPS), there is a growing need for model-driven generation of multi-threaded code and thus mapping on multi-core. We propose a time-predictable multi-core architecture model in architecture analysis and design language (AADL), and map the multi-threaded code to this model.

Keywords synchronous languages, SIGNAL, guarded actions, verified compiler, Coq, architecture analysis and design language (AADL)

Received August 14, 2014; accepted August 2, 2015

E-mail: hukai@buaa.edu.cn

1 Introduction

Safety-critical real-time systems such as avionics, space systems and nuclear power plants, are considered as *reactive systems*, because they always interact with their environment continuously. The environment can be some physical devices to be controlled, a human operator, or other reactive systems. These systems receive from the environment input events, and compute the output information, which is eventually sent to the environment. The synchronous approach is an important choice for the design of these systems, which relies on the *synchronous hypothesis* [1]: a synchronous program reacts to its environment in a sequence of discrete instants. At each instant, the system does input-computation/communication-output, which takes zero time. Even if the physical time is abstracted, the inherent functional properties are not changed, so we can say this method focuses on functional behaviors at a platform-independent level. In contrast to asynchronous concurrency, synchronous languages avoid the introduction of nondeterminism by interleaving. Namely, the execution of two independent, atomic parallel tasks is simultaneous. This allows deterministic semantics, thereby making synchronous programming amenable to predictable system design.

There are several synchronous languages, such as ESTEREL [2], LUSTRE [3], and QUARTZ [4] based on the *perfect synchrony* paradigm, and SIGNAL [5] based on the *polychrony* paradigm. Synchronous languages can be considered as different implementations of the synchronous hypothesis. As a main difference from other synchronous languages, SIGNAL naturally considers a mathematical time model, in term of a partial order relation, to describe multi-clocked systems without the necessity of a global clock. This feature permits the description of globally asynchronous locally synchronous systems (GALS) conveniently, where components based on different clock domains are integrated at the system level.

This paper reports a new compiler prototype for the SIGNAL language, including sequential code generation and multi-threaded code generation with time-predictable properties.

1) Intermediate representation

Guarded commands [6], also called *asynchronous guarded actions* by Brandt et al. [7], are a well-established concept for the description of concurrent systems. In the spirit of the guarded commands, Brandt et al. propose synchronous guarded actions [8] as an intermediate representation for their QUARTZ compiler. As the name suggests, it follows the synchronous model. Hence, the behavior (control flow as well as data flow) is basically described by sets of guarded actions of the form $\langle \gamma \Rightarrow \mathcal{A} \rangle$. The boolean condition γ is called the guard and $\mathcal{A}$ is called the action. To support the integration of synchronous, polychronous and asynchronous models (such as CAOS [9] or SHIM [10]), they propose an extended intermediate representation, that is, *clocked guarded actions* [7,11] where one can declare explicitly a set of clocks. They also show how clocked guarded actions can be used for verification by symbolic model checking (SMV) and simulation by SystemC.

Compared with the existing SIGNAL compiler-Polychrony[1], we use clocked guarded actions as the intermediate representation, to integrate more synchronous languages such as QUARTZ, AIF[2] [7] into our compiler prototype in the future. However, in contrast to the SIGNAL language, clocked guarded actions can evaluate a variable even if its clock does not hold. We also mention that the DC+ [12] intermediate format has been proposed as an intermediate format for compiling multiclock synchronous languages (ESTEREL, LUSTRE and SIGNAL). However, DC+ is introduced as a layer on top of DC which is a monoclock intermediate language. DC+ is characterized by a rich kernel with a monoclock guarded assignment (named *at*) and the equivalent of SIGNAL *when* and *default* constructs. Thus, we propose a variant of Clocked Guarded Actions, namely S-CGA, which constrains variable accesses as done by SIGNAL, and guarded assignments are multiclocked. Compared to DC+, the SIGNAL *when* and *default* are not part of S-CGA, actually, they are compiled. More generally, to conform with the revised semantics of clocked guarded actions, we also do some adjustments on the translation rules from SIGNAL to clocked guarded actions (which are given by [7,11]).

2) Code generation

The sequential code generation from SIGNAL programs is adapted to the S-CGA context. We also consider enhancements of the compiler and their insertion in the compilation chain. Moreover, we propose an appropriate modular architecture for our prototype.

With the advent of multi-core processors, automated synthesis of multi-threaded code from polychronous models is an attractive option for embedded system designers [13–17]. However, we would like to consider the multi-threaded code generation with time-predictable properties. Time predictability means that the program timing can be foreseen statically, such as worst-case execution time (WCET). In order to measure WCET in a compositional way, strong architectural hypotheses must be done, and this is the goal of the time-predictable architecture. In this paper, we propose a time-predictable multi-core architecture model in architecture analysis and design language (AADL) [18], and then we map the multi-threaded code to this model.

3) Verification of the compilation

For a safety-critical system, it is naturally required that the compiler must be verified to ensure that the source program semantics is preserved. There are many approaches to gain assurance that the transformation or the translation of a specification or a program is semantic-preserving. This can be done by directly building a theorem-prover-verified compiler [19], by using translation validation [20], etc. The existing formal verification techniques around SIGNAL are mainly based on translation validation [20,21]. However, translation validation treats the compiler as a "black box", namely it just checks the input and output of each program transformation to validate the semantics preservation, so it yields that one needs to redo the validation when the source program is changed. We would like to extract a verified SIGNAL compiler from a correctness proof developed within the theorem prover Coq, as it has been done in the GENEAUTO project for a part of the SIMULINK compiler [22].

[1] http://www.irisa.fr/espresso/Polychrony
[2] Averest Intermediate Format

Firstly, formal semantics is an important basis for the compiler verification. There exist several semantics for SIGNAL, such as denotational semantics based on traces (called trace semantics) [23–25], denotational semantics based on tags which puts forward a partial order view of time (called tagged model semantics) [24,26], structural operational semantics defining inductively a set of possible transitions [5,24], operational semantics defined by synchronous transition systems (STS) [20]. In Ref. [27], we have studied the equivalence between the trace semantics and the tagged model semantics, to assert a determined and precise semantics of the SIGNAL language.

Secondly, verifying a compiler is always a long-term work. The front-end of our compiler prototype has been proven. However, there already exists a mechanized semantics of a subset of C language in Coq [19], and we have already worked on the mechanized semantics of different AADL subsets such as [28]. Thus we can envision to validate semantically the mapping from the S-CGA level to the targets such as sequential code in C and multi-threaded code in AADL.

To summarize, the relation between our work and related work is shown in Fig. 1 (which extends the figure given in Ref. [7]).

The rest of this paper is structured as follows. Section 2 introduces the basic concepts of the SIGNAL language. The abstract syntax of SIGNAL and its denotational semantics based on the trace model are also given. Section 3 presents the abstract syntax and the denotational semantics of S-CGA. Section 4 gives the front-end of the compiler, i.e., the translation from SIGNAL to S-CGA. The proof of the semantics preservation of the transformation is also presented in Section 4. Section 5 and Section 6 present the sequential code generation and the multi-threaded code generation respectively. Section 7 discusses the related work, and Section 8 gives some concluding remarks.

2 An introduction to SIGNAL

In the SIGNAL language, the variables can be evaluated only at some instants which define their so-called clocks. Moreover, since SIGNAL is polychronous, each variable can have its own clock. In this section, we first introduce the basic concepts and the abstract syntax of the SIGNAL language, and then we present the semantics domain and the trace semantics of SIGNAL.

2.1 Basic concepts and abstract syntax of SIGNAL

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

Example 1 Three signals named $input_1$, $input_2$, $output$ are shown as follows. Here a logical time reference is denoted as

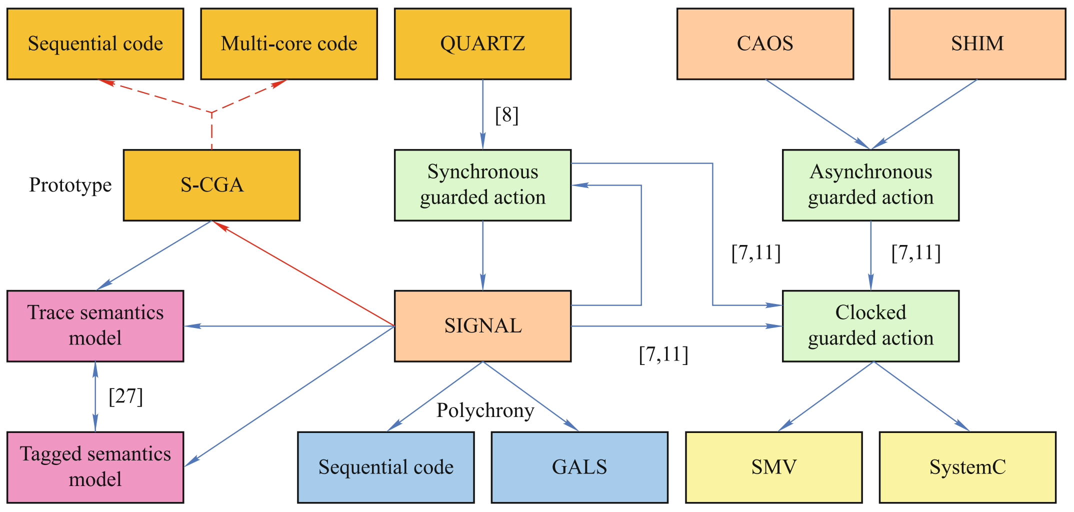


Fig. 1 A global view of the relation between our work and related work

$(t_k)_{k\in\mathbb{N}}$.

$$\begin{array}{lccccc} t & t_1 & t_2 & t_3 & t_4 & \cdots \\ input_1 & 1 & \perp & 3 & \perp & \cdots \\ input_2 & \perp & 5 & 7 & 9 & \cdots \\ output & \perp & \perp & 10 & \perp & \cdots \end{array}$$

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

In the example given above, the abstract clocks of $input_1$, $input_2$ and $output$, denoted respectively $\widehat{input}_1$, $\widehat{input}_2$, and $\widehat{output}$, are defined by different sets of logical instants. For instance, the abstract clock associated with $input_1$ is the set $\{t_1, t_3, ...\}$.

Moreover, SIGNAL can specify the relations between the abstract clocks of signals in two ways: implicitly or explicitly.

• **Primitive constructs** SIGNAL uses several primitive constructs to express the relations between signals, including relations between values, and relations between abstract clocks. Moreover, the primitive constructs can be classified into two families: monoclock operators (for which all signals involved have the same abstract clock) and multiclock operators (for which the signals involved may have different clocks).

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

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

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

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

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

The semantics of each of the extended constructs is defined in term of the primitive constructs [24], so we just consider the primitive constructs, that is kernel SIGNAL (kSIGNAL for short). Its abstract syntax is presented as follows:

$$\begin{array}{lll} P ::= & x := f(x_1, x_2, \dots, x_n) & \text{(instantaneous function)} \\ & |x := x_1 \ \$ \ init\ c & \text{(delay)} \\ & |x := x_1\ when\ x_2 & \text{(undersampling)} \\ & |x := x_1\ default\ x_2 & \text{(deterministic merging)} \\ & |P|P' & \text{(composition).} \end{array}$$

We can use both primitive constructs and extended constructs in the programming. However, the compiler will translate it into kSIGNAL (just use primitive constructs). Thus, in the proof of semantics preservation, we consider kSIGNAL and S-CGA.

In order to get the simplest criterion for the proof of semantics equivalence, local variables are supposed to be moved to the top level, so that the corresponding signals can be controlled from the outside. It means that non-deterministic behaviors are excluded, but our goal is to generate executable code, not specifications.

2.2 Trace model

There exist several semantics for SIGNAL, such as trace semantics (which is used in the reference manual for SIGNAL Version 4), tagged model semantics (based on tags which puts forward a partial order view of time), structural operational semantics. This paper considers the trace semantics. In the following paragraphs, we summarize the trace model [23,34] which is used to define the trace semantics of SIGNAL.

Let $\mathcal{X}$ be a set of variables, and let $\mathcal{V}$ be the set of values that can be taken by the variables. For a variable $x \in \mathcal{X}$, and a non-empty subset X of variables in $\mathcal{X}$, we consider $\mathcal{V}_x$ the domain of values that may be taken by x, and $\mathcal{V}_X = \bigcup_{x \in X} \mathcal{V}_x$.

The symbol $\perp$ ($\perp \notin \mathcal{V}$) is introduced to express the absence of valuation of a variable. Then we denote:

$$\mathcal{V}^{\perp} = \mathcal{V} \cup \{\perp\},$$

$$\mathcal{V}_X^{\perp} = \mathcal{V}_X \cup \{\perp\}.$$

The basic objects manipulated by the SIGNAL language are signals. The length of a signal can be either finite or infinite.

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

The definition of a trace will be given later. Note that, a signal is just a sequence of values corresponding to a signal variable, while a trace defines the synchronized sequences of values of a set of signal variables.

Definition 2 (event) Considering X a non-empty subset of $\mathcal{X}$, we call event on X any application

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

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

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

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

Definition 3 (traces) Φ_X is the set of traces on $\mathcal{X}$, defined as the set of applications $\mathbb{N} \to \mathcal{E}_X$ where $\mathbb{N}$ is the set of natural numbers.

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

The absent trace on X ($\mathbb{N} \to \{\perp_e(X)\}$), i.e., the infinite sequence formed by the infinite repetition of $\perp_e(X)$, is denoted $\perp_X$.

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

Definition 5 (trace equivalence) Two traces are equivalent, if and only if they have the same set of signal variables and the same set of signals.

2.3 Trace semantics of SIGNAL

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

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

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

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

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

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

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

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

$$\forall t \in \mathbb{N},$$
$$x_t = \begin{cases} x_{1t}, & \text{if } x_{2t} = \text{true}; \\ \perp, & \text{otherwise}. \end{cases}$$

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

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

$$\forall t \in \mathbb{N},\quad x_t = \begin{cases} x_{1t}, & \text{if } x_{1t} \neq \perp; \\ x_{2t}, & \text{otherwise.} \end{cases}$$

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

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

3 Synchronous clocked guarded actions for SIGNAL

In papers such as [11], clocked guarded actions have been defined as a common representation for synchronous (via synchronous guarded actions), polychronous and asynchronous (via asynchronous guarded actions) models. It has a multi-clocked feature. However, in contrast to the SIGNAL language, clocked guarded actions can evaluate a variable even if its clock does not hold [11,29], in this case the read value is the most recently written value, while in SIGNAL read and writes can be simultaneous provided the causality is respected. As a consequence, we introduce S-CGA, which is a variant of clocked guarded actions. S-CGA constrains variable accesses as done by SIGNAL. We remark that the SIGNAL compiler has introduced intermediate representations to manage guards and dependencies such as hierarchized conditional dependency graph (HCDG) [26]. The proposed intermediate language is not at the same level: S-CGA does not resolve dependencies. Actually, HCDG could be reused in the next stages of the compilation process. In this section, we first present the syntax of S-CGA, and then we give the denotational semantics of S-CGA based on the trace model.

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

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

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

where

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

An immediate assignment $x = \tau$ writes the value of τ immediately to the variable x. The form (1) implicitly imposes that if γ is defined[3)] and its value is true, then x is present and τ is defined.

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

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

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

Definition 7 (Trace semantics of S-CGA) The trace semantics of an S-CGA system is defined as a set of traces, that is $[\![SCGA]\!] = \{S \mid \forall scga \in SCGA, [\![scga]\!]_S = true\}$. We have the following semantics rules,

$$\begin{aligned} (1)\ & [\![\gamma \Rightarrow x = \tau]\!]_S = \\ & \forall t \in \mathbb{N}, \widehat{[\![\gamma]\!]}_{S,t} \wedge [\![\gamma]\!]_{S,t} \\ & \rightarrow (\widehat{[\![x]\!]}_{S,t} \wedge \widehat{[\![\tau]\!]}_{S,t} \wedge [\![x]\!]_{S,t} = [\![\tau]\!]_{S,t}), \end{aligned}$$

$$\begin{aligned} (2)\ & [\![\gamma \Rightarrow next(x) = \tau]\!]_S = \\ & \forall t_1 < t_2 \in \mathbb{N}, \\ & ((\forall t' \in \mathbb{N},\ t_1 < t' < t_2 \rightarrow \neg\widehat{[\![x]\!]}_{S,t'}) \wedge \widehat{[\![\gamma]\!]}_{S,t_1} \wedge [\![\gamma]\!]_{S,t_1}) \\ & \rightarrow (\widehat{[\![x]\!]}_{S,t_1} \wedge \widehat{[\![\tau]\!]}_{S,t_1} \wedge (\widehat{[\![x]\!]}_{S,t_2} \rightarrow [\![x]\!]_{S,t_2} = [\![\tau]\!]_{S,t_1})), \end{aligned}$$

$$\begin{aligned} (3)\ & [\![\gamma \Rightarrow\ assume(\sigma)]\!]_S = \\ & \forall t \in \mathbb{N}, \widehat{[\![\gamma]\!]}_{S,t} \wedge [\![\gamma]\!]_{S,t} \rightarrow \widehat{[\![\sigma]\!]}_{S,t} \wedge [\![\sigma]\!]_{S,t}, \end{aligned}$$

3) An expression is said to be defined if all the variables it contains are present

where $\widehat{[\![e]\!]}_{S,t}$ defines the domain of e: it is true if all the variables of e are present in trace S at the instant t; $[\![e]\!]_{S,t}$ is a partial function defined over the domain $\widehat{[\![e]\!]}_{S,t}$ whose value is the valuation of e on trace S at the instant t.

- Rule (1): when γ is present, and the value of γ is true, x and τ are both present, and the value of x is that of τ.
- Rule (2): when γ is present and the value of γ is true at instant t_1, x and τ are present at t_1, and if t_2 is the next instant where x is present, then the value of x at t_2 is that of τ at instant t_1.
- Rule (3): when γ is present, and the value of γ is true, σ is present and true.

The semantics of S-CGA composition is defined as $[\![scga_1 \parallel scga_2]\!]_S = [\![scga_1]\!]_S \wedge [\![scga_2]\!]_S$.

4 From kSIGNAL to S-CGA and its semantics preservation

In this section, we present the front-end of the compiler, that is, the translation from kSIGNAL to S-CGA. We envision the extraction of a complete verified-compiler prototype from the theorem proof. Thus, we would like to use the theorem prover Coq, to express and verify the translation from kSIGNAL to S-CGA.

4.1 Translation rules

kSIGNAL can be structurally translated to S-CGA by translating each construct separately. The translation rules are close to the ones which have been given in Ref. [7]. However, to conform with the semantics of S-CGA (i.e., the revised semantics of clocked guarded actions), we have done some adjustments.

kSIGNAL		S-CGA
(1) $x := f(x_1, \ldots, x_n)$	$\Rightarrow$	$\begin{cases} \hat{x} \Rightarrow x = f(x_1, \ldots, x_n) \\ \parallel \widehat{x_1} \Rightarrow assume(\hat{x}) \\ \parallel \ldots \\ \parallel \widehat{x_n} \Rightarrow assume(\hat{x}) \end{cases}$
(2) $x := x_1 \ \$ \ init\ c$	$\Rightarrow$	$\begin{cases} init(\hat{x}) \Rightarrow x = c \\ \parallel \hat{x} \Rightarrow next(x) = x_1 \\ \parallel true \Rightarrow assume(\hat{x} = \widehat{x_1}) \end{cases}$
(3) $x := x_1\ when\ x_2$	$\Rightarrow$	$\begin{cases} \widehat{x_1} \wedge x_2 \Rightarrow x = x_1 \\ \parallel \hat{x} \Rightarrow assume(\widehat{x_1} \wedge x_2) \end{cases}$
(4) $x := x_1\ default\ x_2$	$\Rightarrow$	$\begin{cases} \widehat{x_1} \Rightarrow x = x_1 \\ \parallel \widehat{x_2} \wedge \neg\widehat{x_1} \Rightarrow x = x_2 \\ \parallel \hat{x} \Rightarrow assume(\widehat{x_1} \vee \widehat{x_2}) \end{cases}$

- Translation (1): The *instantaneous function* is applied to the inputs and produces the output. Note that the immediate assignment $\hat{x} \Rightarrow x = f(x_1, \ldots, x_n)$ implicitly imposes $\hat{x} \Rightarrow \widehat{x_1}, \ldots, \hat{x} \Rightarrow \widehat{x_n}$, so in the assumption we only assert $\widehat{x_1} \Rightarrow assume(\hat{x}), \ldots, \widehat{x_n} \Rightarrow assume(\hat{x})$. Thus all variables have the same clock as required by the semantics of SIGNAL.
- Translation (2): The translation of the *delay* construct is split up in two cases. a) The first value that is produced by this construct is the constant c at the first instant when x is present. b) In all other instants, the value of x is assigned by the value of x_1 evaluated at the last non-absent instant of $\widehat{x_1}$. The assumption ensures that both variables have the same clock.
- Translation (3): The *undersampling* construct transfers the value of x_1 to x whenever it is needed. The clock assumption ensures that $\hat{x}$ only holds when both inputs (i.e., x_1 and x_2) are present and x_2 is true. Thanks to the assume semantics (Rule(3) of Definition 7), $assume(\widehat{x_1} \wedge x_2)$ implies $\widehat{x_1} \wedge \widehat{x_2} \wedge x_2$.
- Translation (4): The *deterministic merging* construct merges two signals with priority for the first one. Therefore, if the first input is present, it is passed to x. If it is not present, but the second one is, then the second one is passed to x. The clock assumption ensures that $\hat{x}$ only holds when at least one of the inputs is present.

Remark Compared with the translation rules given in Ref. [7], the main change is in the Translation (3). Namely, $true \Rightarrow assume(\hat{x} = \widehat{x_1} \wedge \widehat{x_2} \wedge x_2)$ has been changed into $\hat{x} \Rightarrow assume(\widehat{x_1} \wedge x_2)$. According to the Rule (3) (Definition 7): when γ is present, and the value of γ is true, σ must be present, and the value of σ is true. $true \Rightarrow assume(\hat{x} = \widehat{x_1} \wedge \widehat{x_2} \wedge x_2)$ implies x_2 is always present and always true. Thus, to conform with the semantics of S-CGA, we change it into $\hat{x} \Rightarrow assume(\widehat{x_1} \wedge x_2)$. It means when x is present, $\widehat{x_1} \wedge x_2$ is present and true, i.e., x_1 is present and x_2 is present and true.

Example 2 A translation from kSIGNAL to S-CGA[4] is given as follows.

[4] If two guarded actions update the same variables, the guards must be exclusive

$$
\begin{array}{l}
(|y_1 := x\$\ init\ 1 \\
\ |y_2 := x\$\ init\ 2 \\
\ |z\ := x > 0 \\
\ |s_1 := f(y_1)\ when\ z \\
\ |s_2 := s_1 + 1 \\
\ |s_3 := f(y_2)\ when\ (not\ z) \\
\ |s_4 := s_3 + 2 \\
\ |)
\end{array}
\Rightarrow
\begin{array}{l}
true \Rightarrow assume(\hat{y_1} = \hat{x}) \\
init(\hat{y_1}) \Rightarrow y_1 = 1 \\
\hat{y_1} \Rightarrow next(y_1) = x \\
true \Rightarrow assume(\hat{y_2} = \hat{x}) \\
init(\hat{y_2}) \Rightarrow y_2 = 2 \\
\hat{y_2} \Rightarrow next(y_2) = x \\
true \Rightarrow assume(\hat{x} = \hat{z}) \\
\hat{z} \Rightarrow z = (x > 0) \\
\hat{s_1} \Rightarrow assume(\hat{z} \wedge z) \\
\hat{z} \wedge z \Rightarrow s_1 = f(y_1) \\
\hat{s_2} \Rightarrow s_2 = s_1 + 1 \\
\hat{s_1} \Rightarrow assume(\hat{s_2}) \\
\hat{s_3} \Rightarrow assume(\hat{z} \wedge (not\ z)) \\
\hat{z} \wedge (not\ z) \Rightarrow s_3 = f(y_2) \\
\hat{s_4} \Rightarrow s_4 = s_3 + 2 \\
\hat{s_3} \Rightarrow assume(\hat{s_4})
\end{array}
$$

4.2 The proof of semantics preservation

As shown in Fig. 2, the Coq mechanization includes seven modules (about 1300 lines of Coq code), i.e., the abstract syntax of kSIGNAL, the trace model, the trace semantics of kSIGNAL, the abstract syntax of S-CGA, the trace semantics of S-CGA, the translation rules, and the proof of the semantics preservation. Here, the semantics preservation is defined as a trace equivalence between two trace semantics models related to kSIGNAL and its translation into S-CGA respectively.

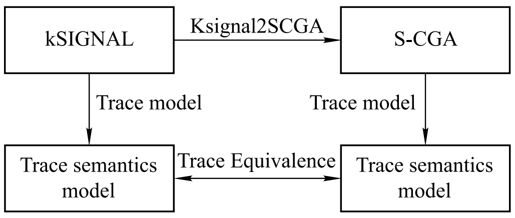


Fig. 2 The global view of the semantics preservation

All the definitions given above have been mechanized in Coq. Here, we just present the main idea of the proof.

Firstly, we prove each semantics rule of the trace semantics of kSIGNAL is trace equivalent with its translation into S-CGA. For each semantics rule, there are two Lemmas to be proven (in two directions).

• **Instantaneous function** 1) Its trace semantics is defined as *Sassignment*. 2) As defined in Section 4.1, its translated guarded actions are $\hat{x} \Rightarrow x = f(x_1, \ldots, x_n)$, $\widehat{x_1} \Rightarrow assume(\hat{x})$, $\ldots$, and $\widehat{x_n} \Rightarrow assume(\hat{x})$. Applying the semantics of S-CGA (*scgaSimm* is the semantics of immediate assignment), we can get the semantics of instantaneous function construct translated into S-CGA. Then, we prove the trace equivalence between 1) and 2).

Lemma 1 signal2scga_ass1: ∀ f x xi tr,
Sassignment x f xi tr →
(scgaSimm ^x x {|exp_fun:=f;exp_args:=xi|} tr
  ^ strModel.straces
  (scga2Sprocess(GA_ipar (fun i: FctAr f
   ⇒ ^(xi i) ⟹ assume(^x))))tr).

Lemma 2 signal2scga_ass2: ∀ f x xi tr,
scgaSimm ^x x {|exp_fun:=f;exp_args:=xi|} tr
→ strModel.straces
  scga2Sprocess (GA_ipar (fun i: FctAr f ⇒
   ^(xi i) ⟹ assume(^x))))tr
→ Sassignment x f xi tr.

• **Delay** 1) Its trace semantics is defined as *Sdelay*. 2) There are three translated guarded actions, i.e., $init(\hat{x}) \Rightarrow x = c$, $\hat{x} \Rightarrow next(x) = x_1$, and $true \Rightarrow assume(\hat{x} = \widehat{x_1})$. Applying the semantics of S-CGA (*getFirst0* is used to get the first instant when x is present, that is $init(\hat{x})$, *scgaSnext* is the semantics of delayed assignment, and *scgaSctr* is the semantics of assumption), we can get the semantics of delay construct translated into S-CGA. Then, we prove the trace equivalence between 1) and 2). In the Lemmas, $\hat{x} = \widehat{x_1}$ is denoted $\hat{x}$ ^= $\widehat{x_1}$ (as clock synchronization operator in SIGNAL).

Lemma 3 signal2scga_delay1: ∀ x x1 v tr,
Sdelay x x1 v tr →
((scgaSimm init(x) x v tr (getFirst0 tr))
^ (∃ c:Value,
  scgaSnext gTrue x x1 c tr (getFirst0 tr)))
^ scgaSct rgTrue (^x ^= ^x1) tr (getFirst0 tr).

Lemma 4 signal2scga_delay2: ∀ x x1 v tr,
scgaSimm init(x) x v tr (getFirst0 tr)
→ (∃ c: Value,
  scgaSnext gTrue x x1 c tr (getFirst0 tr))
→ scgaSctr gTrue (^x ^= ^x1) tr (getFirst0 tr)
→ Sdelay x x1 v tr.

• **Undersampling** 1) Its trace semantics is defined as *Swhen*. 2) There are two translated guarded actions, i.e., $\widehat{x_1} \wedge x_2 \Rightarrow x = x_1$ and $\hat{x} \Rightarrow assume(\widehat{x_1} \wedge x_2)$. Applying the semantics of S-CGA, we can get the semantics of undersampling construct translated into S-CGA. Then, we prove the trace equivalence between 1) and 2). In the Lemmas, $\widehat{x_1} \wedge x_2$ is denoted $\widehat{x_1}$ ^* x_2 (reusing the clock intersection operator of SIGNAL).

Lemma 5 signal2scga_when1: ∀ x x1 x2 tr,
Swhen x x1 x2 tr →
scgaSimm (^x1 ^* x2) x x1 tr
∧ scgaSctr ^x (^x1 ^* x2) tr.

Lemma 6 signal2scga_when2: ∀ x x1 x2 tr,
scgaSimm (^x1 ^* x2) x x1 tr
→ scgaSctr ^x (^x1 ^* x2) tr
→ Swhen x x1 x2 tr.

• **Deterministic merging** 1) Its trace semantics is defined as *Sdefault*. 2) There are three translated guarded actions, i.e., $\widehat{x_1} \Rightarrow x = x_1$, $\widehat{x_2} \wedge \neg\widehat{x_1} \Rightarrow x = x_2$, and $\hat{x} \Rightarrow assume(\widehat{x_1} \vee \widehat{x_2})$. Applying the semantics of S-CGA, we can get the semantics of deterministic merging construct translated into S-CGA. Then, we prove the trace equivalence between 1) and 2). In the Lemmas, $\widehat{x_2} \wedge \neg\widehat{x_1}$ is denoted $\widehat{x_2}$ ^− $\widehat{x_1}$ (clock difference operator of SIGNAL), and $\widehat{x_1} \vee \widehat{x_2}$ is denoted $\widehat{x_1}$ ^+ $\widehat{x_2}$ (clock union operator of SIGNAL).

Lemma 7 signal2scga_default1: ∀ x x1 x2 tr,
Sdefault x x1 x2 tr →
(scgaSimm ^x1 x x1 tr
∧ scgaSimm (^x2 ^– ^x1) x x2 tr)
∧ scgaSctr ^x (^x1 ^+ ^x2) tr.

Lemma 8 signal2scga_default2: ∀ x x1 x2 tr,
scgaSimm ^x1 x x1 tr
→ scgaSimm (^x2 ^– ^x1) x x2 tr
→ scgaSctr ^x (^x1 ^+ ^x2) tr
→ Sdefault x x1 x2 tr.

Secondly, based on these Lemmas, we prove the following Theorem 1, that the two semantics models, i.e., (Process2Sprocess P) and (scga2Sprocess(signal2scga P)) are trace equivalent (they have the same set of signal variables and the same set of traces). This property concerns infinite objects and cannot generally be proved automatically. This is why we use the proof assistant which verifies a user-assisted proof.

```
Record SPeq (p1p2: Sprocess): Prop:=
{
  SPd: ∀ y, sdom p1 y ↔ sdom p2 y;
  SPs: ∀ tr, straces p1 tr ↔ straces p2 tr
}.
```

Theorem 1 signal2scga_check: ∀ p,
SPeq(Process2Sprocess p)
(scga2Sprocess(signal2scga p)).

Finally, we can extract the corresponding CAML code to synthesize the first stage of the verified compiler prototype.

5 Sequential code generation

The compilation process of synchronous languages is not limited to code generation: some analyses are first applied to determine if the specification is indeed executable. The SIGNAL compilation process contains one major analysis called *clock calculus* from which *code generation* directly follows. Moreover the clock calculus contains several steps [26], such as construction of an equation system over clocks, resolution of the system of clock equations, construction of a clock hierarchy on which the automatic code generation strongly relies.

For a safety-critical system, it is important to optimize the control structure of the generated code. In the SIGNAL compiler, the control flow expressed by abstract clocks serves to derive a control structure in automatic code generation. Thus, the quality of clock calculus has a strong impact on the correctness and efficiency of implementations. In Ref. [30], the authors have shown that there is a limitation of the clock calculus of the SIGNAL compiler. For example, for the undersampling construct $x = x_1$ *when* x_2, the clock of the Boolean expression x_2 is partitioned into $[x_2]$ and $[\neg x_2]$, which are referred to as *condition-clocks*. If x_2 is defined by a numerical expression such as an integer comparison, $[x_2]$ and $[\neg x_2]$ are seen as black boxes. This has a strong impact on the analysis precision and the quality of generated code. Thus, the authors propose a new clock abstraction, i.e., *combined numerical-Boolean* abstraction, to eliminate this problem. They also use an SMT solver to reason on the new abstraction. With the same purpose, in Ref. [31], an interval-based data structure referred to as interval-decision diagram (IDD) is considered for the analysis of numerical properties in SIGNAL programs.

As shown in Fig. 3, in our compiler prototype (sequential code generation): 1) We have considered the main components of the clock calculus, such as construction of an equation system over clocks, resolution of a system of clock equations, and construction of a clock hierarchy. 2) To integrate more synchronous languages, such as QUARTZ and AIF, into our prototype in the future, we introduce S-CGA as the intermediate representation, and we adapt the clock calculus to S-CGA. 3) We propose an appropriate modular architecture for our prototype. One benefit of this approach is that we just need to redo a part of proof when some modules of the compilation process are changed. 4) We have considered existing

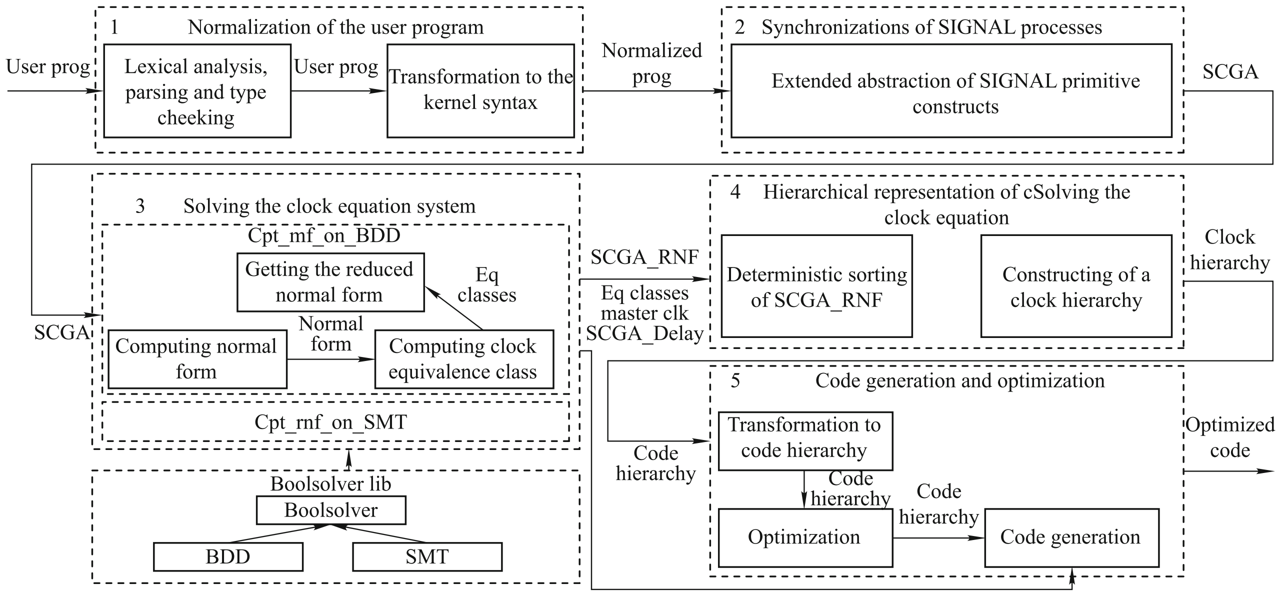

1 Normalization of the user program
User prog
Lexical analysis, parsing and type cheeking
User prog
Transformation to the kernel syntax
Normalized prog
2 Synchronizations of SIGNAL processes
Extended abstraction of SIGNAL primitive constructs
SCGA
3 Solving the clock equation system
Cpt_mf_on_BDD
Getting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

Fig. 3 The architecture of the verified compiler prototype

enhancements such as [30,31], namely we can use both BDD and SMT in the resolution of the system of clock equations.

Specifically, the compilation process is mainly structured as five modules. At each module, there are several submodules.

- Module 1: Beyond the usual lexical analysis, parsing and type checking, the compiler will transform the user program (using the subset of SIGNAL) whose statements all expressed with both primitive constructs and extended constructs to the normalized program (using kSIGNAL) whose statements are all expressed with primitive constructs.
- Module 2: Different from the existing SIGNAL compiler, we construct S-CGA from the normalized program. S-CGA contains control flow (a system of clock equations) as well as data flow. As mentioned above, for the under-sampling construct, the SIGNAL compiler consider the *condition-clocks* $[x_2]$ and $[\neg x_2]$ as black boxes if x_2 is defined by a numerical expression. When x_2 is defined by a complex boolean function, we have $[x_2] = \widehat{x_2} \wedge x_2$ and $[\neg x_2] = \widehat{x_2} \wedge \neg x_2$. Based on this abstraction, we can get more precise clock analysis.
- Module 3: If the system of clock equations contains more than one equation with the same clock, the execution of the generated code will check the same control condition several times, which is inefficient. This is why we need to resolve it. All the clock equations are considered as predicates. We can use BDD or SMT technology to check the equivalence of two predicates, and put the corresponding clock variables into the same equivalence class. We also check the *endochrony* property at this step, namely there is just one master clock.
- Module 4: The code generation is based on both the clock hierarchy and the data dependencies. However, there may be clock-to-data cycles. To reduce these cycles, we first sort all the guarded actions. It will be easier to construct a clock hierarchy based on deterministic sorting, and we consider the sorting as a depth first search (DFS) order.
- Module 5: The basic idea of code generation is the same as that in the SIGNAL compiler. Furthermore, we do some optimizations based on clock inclusions. Given two equations such as $y = x$ when x_1, and $z = x$ when (x_1 and x_2), there is a clock-inclusion relation: $[x_1 \wedge x_2] \rightarrow [x_1]$, i.e., the clock of $[x_1 \wedge x_2]$ is a subset of the clock of $[x_1]$. Consequently, we can do the code optimization illustrated as follows. If control condition x_1 holds, we do not need to check x_1 again in $x_1\&\&x_2$. We just need to check if x_2 holds or not.

```
if (x1){                      if (x1){
   do actions                    do actions
   ...                           ...
   if (x1&&x2){      ⇒          if (x2){
    do actions                    do actions
    ...}                          ...}
}                             }
```

Actually, the first version of the compiler prototype has been implemented in CAML directly. It is a good way to provide a basis for the Coq mechanization of the prototype. Finally, we envision the extraction of a complete prototype from the mechanization.

6 Multi-threaded code generation with time-predictable properties

Safety-critical embedded systems or cyber-physical systems (CPS) distinguish themselves from general purpose computing systems by several characteristics, such as failure to meet deadlines may cause a catastrophic or at least highly undesirable system failure. Time-predictable system design [32–34] is concerned with the challenge of building systems in such a way that timing requirements can be guaranteed from the design. This means we can predict the system timing statically. The widespread advent of multi-core processors further aggravates the complexity of timing analysis. We would like to consider the multi-threaded code generation with time-predictable properties. At the modeling level, synchronous programming is a good choice for time-predictable system design. At the compiler level, we give the verified compiler from SIGNAL to our intermediate representation S-CGA and thus to multi-threaded code. At the platform level, we propose a time-predictable multi-core architecture model in architecture analysis and design language (AADL) [18], and then we map the multi-threaded code to this model. Therefore, our method integrates time predictability across several design layers.

6.1 From S-CGA to multi-threaded code

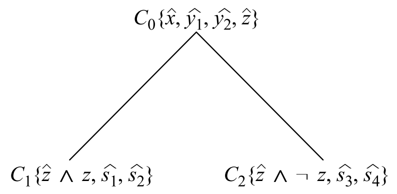
The SIGNAL compilation process contains one major analysis called *clock calculus* from which code generation directly follows. Moreover, the clock calculus contains several steps [26], such as construction of an equation system of relations over clocks, resolution of the system of clock equations, and construction of a clock hierarchy. Our goal here is to adapt the clock calculus to S-CGA. Moreover, in the multi-threaded code generation scheme, the data-dependency graph (DDG) should also be constructed to find more parallelism.

Based on the semantics of S-CGA, we can get the equation system over clocks. The general rules are given as follows.

S-CGA	Clock equations
$\gamma \Rightarrow x = \tau$	$\hat{\gamma} \wedge \gamma \rightarrow \hat{x} \wedge \hat{\tau}$
$\gamma \Rightarrow next(x) = \tau$	$\hat{\gamma} \wedge \gamma \rightarrow \hat{x} \wedge \hat{\tau}$
$\gamma \Rightarrow assume(\sigma)$	$\hat{\gamma} \wedge \gamma \rightarrow \hat{\sigma} \wedge \sigma$
	$init(\hat{x}) \rightarrow \hat{x}\ (\forall x \in X)$

[5] Weak endochrony [35] property will be considered in the future

As the first step, we just consider *endochrony* property[5], namely we can construct a clock hierarchy based on the resolution of the system of clock equations. The clock hierarchy of Example 2 (with three clock equivalence classes, i.e., C_0, C_1, and C_2) is shown in Fig. 4. For instance, the signals x, y_1, y_2, and z are synchronous, thus they are in the same clock equivalence class (C_0).



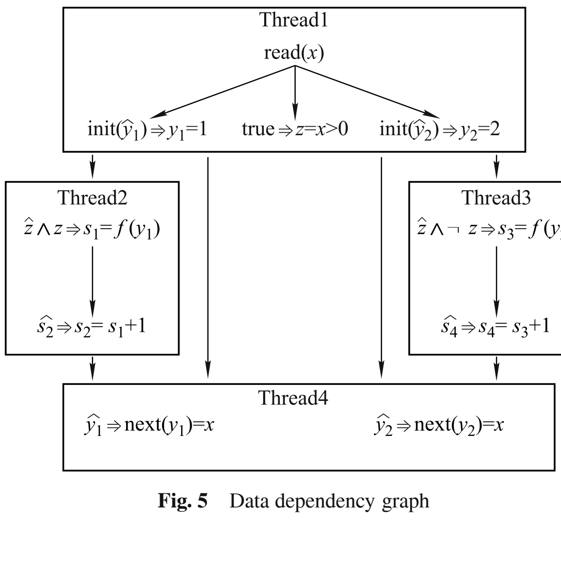
$C_0\{\hat{x}, \hat{y_1}, \hat{y_2}, \hat{z}\}$
$C_1\{\hat{z} \wedge z, \hat{s_1}, \hat{s_2}\}$
$C_2\{\hat{z} \wedge \neg z, \hat{s_3}, \hat{s_4}\}$

Fig. 4 Clock hierarchy

The properties of the clock hierarchy are presented as follows [36].

- Each node is a clock equivalence class.
- There is just one master clock equivalence class, here is C_0 in the clock hierarchy.
- There is a clock implication relation (checked by BDD or SMT) between a son node and its father node, for instance, $\hat{z} \wedge z \rightarrow \hat{z}$. Thus, all the clocks in the clock hierarchy can be defined from the master clock.

In the sequential code generation scheme, we associate guarded actions to each clock equivalence class of the clock hierarchy, then the deterministic sequential code will be generated [29]. In the multi-threaded code generation schema, the DDG should also be constructed to find more parallelism. We construct the DDG, as shown in Fig. 5, based on the variables reading and writing.



Thread1
read(x)
init($\hat{y}_1$)$\Rightarrow y_1$=1
true$\Rightarrow z$=x>0
init($\hat{y}_2$)$\Rightarrow y_2$=2
Thread2
$\hat{z}\wedge z\Rightarrow s_1=f(y_1)$
$\hat{s}_2\Rightarrow s_2=s_1+1$
Thread3
$\hat{z}\wedge\neg z\Rightarrow s_3=f(y_2)$
$\hat{s}_4\Rightarrow s_4=s_3+1$
Thread4
$\hat{y}_1\Rightarrow$next(y_1)=x
$\hat{y}_2\Rightarrow$next(y_2)=x

Fig. 5 Data dependency graph

Definition 8 (reading and writing dependencies) [37] Let $FV(\tau)$ denote the free variables occurring in the expression τ. The dependencies from guarded actions to variables are defined as follows:

$$RdVars(\gamma \Rightarrow x = \tau) := FV(\gamma) \cup FV(\tau);$$
$$RdVars(\gamma \Rightarrow next(x) = \tau) := FV(\gamma) \cup FV(\tau);$$
$$WrVars(\gamma \Rightarrow x = \tau) := \{x\};$$
$$WrVars(\gamma \Rightarrow next(x) = \tau) := \{next(x)\}.$$

Then, the dependencies from variables to guarded actions are defined as follows:

$$RdActs(x) := \{\gamma \Rightarrow \mathcal{A} \mid x \in RdVars(\gamma \Rightarrow \mathcal{A})\};$$
$$WrActs(x) := \{\gamma \Rightarrow \mathcal{A} \mid x \in WrVars(\gamma \Rightarrow \mathcal{A})\}.$$

Note that, $\gamma \Rightarrow assume(\sigma)$ is used in the construction of the clock hierarchy. We expect that it is not needed to associate some actions to *assume* inside the DDG. This leads to proof obligations that should be checked to guarantee the correctness of the generated code.

An action can only be executed if all read variables are known. Similarly, a variable is only known if all actions writing it in the current step have been evaluated before.

The multi-threaded code generation depends on the data dependency graph which has been associated with the clock hierarchy. We first need to find partitions. As presented in Fig. 6, we would like to treat the partition methods in general, which means different partition methods (such as the vertical way [38] for a concurrent execution and the horizontal way [39] for a pipelined execution) do not affect the proof. Our approach is general and it only requires a legal partition, here we reuse the definition of [37].

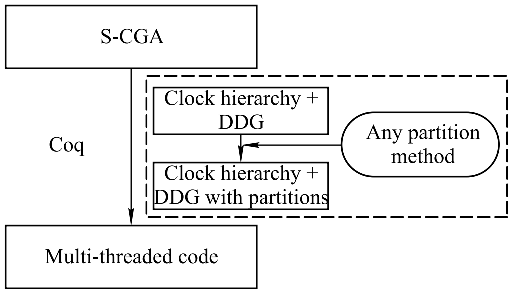


Fig. 6 The proof idea

Definition 9 (legal partition) Let P be a partition, A_1 and A_2 be guarded actions of P, and $\sqsubseteq$ be the reflexive and transitive closure of the following relation $R \subseteq \mathcal{A} \times \mathcal{A} : (A_1, A_2) \in R \Leftrightarrow WrVars(A_1) \cap RdVars(A_2) \neq \{\}$. P is legal if and only if $\sqsubseteq$ is a partial order.

[6)] Time-predictable Multi-Core Architecture for Embedded Systems

Note that, the intersection of $WrVars(A_1)$ and $RdVars(A_2)$ is empty, if A_1 is a delayed action for a reading variable in A_2.

Based on the Definition 9, a partition scheme of Example 2 (with four partitions) is given in Fig. 5. The basic principle of our partition method is described as follows.

- Consider one partition (i.e., one thread) for each vertex of the data-dependency graph.
- Merge two partitions for example P_1 and P_2, if P_2 is the only son of P_1, and P_1 is the only father of P_2.
- In each partition, we organize the guarded actions based on the clock equivalence classes. For example, the two guards in *Thread2* belong to the same clock equivalence class, so they will be merged inside the same control condition in the generated code.

Finally, we add wait/notify synchronization among the threads. A code fragment of Thread2 is given as follows.

```
/ * Thread 2 * /
void step()
{
 wait(Thread1);
 if(C1){
    s1 = f(y1);
    s2 = s1 + 1; }
 notify(Thread4);
}
```

6.2 Mapping multi-threaded code to multi-core

To allow for static prediction of the system timing, we need time-predictable processor architectures, thus we know all the architecture details such as the pipeline and the memory hierarchy to analyze the execution time of programs. Furthermore, the mapping from multi-threaded code to multi-core architectures should be also static and deterministic.

6.2.1 A time-predictable multi-core architecture model

With the advent of multi-core architectures, interference between threads on shared resources further complicates analysis. There are some recommendations from Wilhelm et al. [33,34], i.e., the better way is to reduce the time interference: (1) pipeline with static branch prediction and with in-order execution; (2) separation of caches (instruction and data caches); (3) LRU (Least Recently Used) cache replace policy; and (4) access of main memory via a time division multiple access (TDMA) scheme. In the EC funded project T-CREST[6)], Schoeberl et al. [40, 41] propose a new form of organization for the instruction cache, named method cache (MC). They split data caches (including stack cache (SC), static data cache (SDC), constants data cache (CDC), and heap allocated data cache (HC)), to increase the time predictability and to tighten the WCET. The method cache stores complete methods, and cache misses occur only on method invocation and return. They split the data cache for different data areas, thus data cache analysis can be performed individually for the different areas. In our work, heap is avoided to be used because we do not use dynamic memory allocation in our multi-threaded code.

Based on the existing work, we would like to model a time-predictable multi-core architecture in AADL. AADL is an SAE (society of automotive engineers) architecture description language standard for embedded real-time systems, and supports several kinds of system analysis such as schedulability analysis. Moreover, we have already worked on the semantics of different AADL subsets such as Ref. [28]. So we envision how to validate semantically the mapping from the language level to the architecture level.

Our multi-core architecture model is illustrated in Fig. 7. Inside the core, we consider static branch prediction and in-order execution in the pipeline. A simplified instruction set (*get_instruction*, *compute*, *write_data*, and *read_data*) is used. As the first step, we just consider first level cache (i.e., without L2 and L3). Each core is associated with a method cache, a stack cache, a static data cache, and a constants data cache. However, the same principle of cache splitting can be applied to L2 and L3 caches. The extension of the timing analysis for a cache hierarchy is straightforward. Moreover, TDMA-based resource arbitration allocates statically-computed slots to the cores.

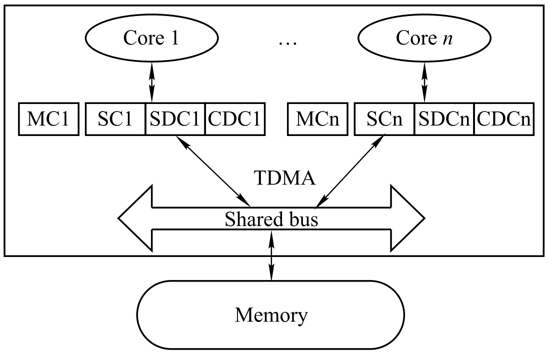


Fig. 7 A time-predictable multi-core architecture model

As proposed by Ref. [42], a core is associated with an AADL *processor* component and a multi-core processor with an AADL *system* component containing multiple AADL processor subcomponents, each one representing a separate core. This modeling approach provides flexibility: an AADL system can contain other components to represent cache, and shared bus, etc. For that purpose, we define specific modeling patterns with new properties such as *MC_Properties*. *TDMA_Window* denotes a *Slot* to an *access connection*. For a bus, there will be a list of allocations of slots, that is *TDMA_Schedule*. For instance, for *N* access connections, the TDMA period of the bus is *T=Slot * N*. Here, we consider all the access connections have the same slot duration.

```
property set MC_Properties is
 TDMA_Window : type record (
   AccessPoint : list of reference (access connection);
   Slot : time;
   );
 TDMA_Schedule : list of MC_Properties
                        :: TDMA_Window applies to (bus);
...
 end MC_Properties;
```

6.2.2 The mapping method

To preserve the time predictability, we consider static mapping and scheduling. Example 2 generates a configuration file (such as *num_of_threads*=4) in multi-threaded code generation. Moreover, we have a manual configuration file for the time-predictable multi-core architecture model, for example *num_of_cores*=4. Thus, we can generate a static mapping and scheduling, for instance:

- Thread1 $\mapsto$ Core1, Thread2 $\mapsto$ Core2, Thread3 $\mapsto$ Core3, and Thread4 $\mapsto$ Core4.
- Thread1: notify(Thread2), notify(Thread3);
  Thread2: wait(Thread1), notify(Thread4);
  Thread3: wait(Thread1), notify(Thread4);
  Thread4: wait(Thread2), wait(Thread3).

Thanks to the mechanizations such as method cache, split data caches, TDMA and static scheduling, the execution time of the multi-threaded code can be bounded.

7 Related work

In this section, we discuss some related work about two aspects: verification of the SIGNAL compilation (mainly focuses on sequential code generation) and multi-threaded code generation from SIGNAL.

7.1 Verification of the SIGNAL compilation

For a safety-critical system, it is naturally required that the compiler must be verified to ensure that the source program semantics is preserved. For example, the SCADE Suite KCG automatic C code generator has been qualified as a development tool at DO-178B level A. Moreover, one of the supplements to DO-178C, the DO-330 (software tool qualification considerations), provides a guidance to qualify tools. This means a tool, e.g., a development tool or a verification tool, also needs to be qualified. There are many approaches to gain assurance that the transformation or the translation of a specification or a program is semantic-preserving. This can be done by directly building a theorem-prover-verified compiler [19], by using translation validation [20], etc.

Pnueli et al. [20] propose the notion of translation validation to verify the code generator of SIGNAL. In that work, the authors define a language of symbolic models to represent both the source and target programs, called synchronous transition systems (STS). An STS is a set of logic formulas which describe the functional and temporal constraints of the whole program and its generated C code. Then they use BDD representations to implement the symbolic STS models, and their proof method uses a SAT-solver to reason on the signal constraints.

In Ref. [21], the authors adopt translation validation to formally verify that the clock semantics and data dependence are preserved during the compilation of the SIGNAL compiler. They represent the clock semantics, the data dependence of a program and its transformed counterpart as first-order formulas which are called Clock Models and synchronous dependence graphs (SDGs) respectively. Then they introduce clock refinement and dependence refinement relations which express the preservations of clock semantics and dependence, as a relation on clock models and SDGs respectively. Finally, an SMT-solver is used for checking the existence of the correct transformation relations.

In Ref. [43], the authors encode the source SIGNAL programs and their transformations with polynomial dynamical systems (PDSs), and prove that the transformations preserve the abstract clocks and clock relations of the source programs. By using the simulation in model checking techniques, their approach suffers from the increasing of the state-space when it deals with large programs.

These existing researches mainly use the method of translation validation. However, translation validation treats the compiler as a “black box”, namely it just checks the input and output of each program transformation to validate the semantics preservation. Therefore, it yields that one needs to redo the validation when the source program is changed. We would like to extract a verified SIGNAL compiler which considers a subset of the SIGNAL language, based on the theorem-prover-verified compiler method [19]. Moreover, the challenge is to be modular enough to make proof compositional, and to be able to update the proof when we need to do more optimization.

7.2 Multi-threaded code generation from SIGNAL

The report [13] describes all code generation strategies available in the Polychrony toolset. In the multi-threaded code generation scheme, it uses micro-level threading which creates a large number of threads and equally large number of semaphores, leading to inefficiency. Thus, Jose et al. [14] propose a process-oriented and non-invasive multi-threaded code generation using the sequential code generators. It means that instead of changing the compiler, they use the existing sequential code generator and separately synthesize some programming glue to generate multi-threaded code. Papailiopoulou et al. [16] define a full design flow starting from high level domain specific languages (e.g., Simulink, SCADE, AADL, SysML, MARTE, and SystemC), transforming to polychronous specifications, and going to the generation of deterministic concurrent (multi-threaded) executable code for simulation or (possibly distributed) implementation. The multi-threaded code generation in Refs. [14] and [16] are both based on the *weak endochrony* property.

There are also some work about multi-threaded code generation from the guarded actions. Baudisch et al. [38] present a compilation of synchronous programs to multi-threaded OpenMP-based C programs. They start at the level of synchronous guarded actions. In addition to the explicit parallelism given in the source program, their method also exploits the implicit parallelism which is due to the underlying synchronous model of computation and the data dependencies of the guarded actions. To speedup the execution of multi-threaded code, Baudisch et al. [39] propose an automatic synthesis procedure that translates synchronous guarded actions to software pipelines. The synchronous guarded actions are analyzed in terms of their data-dependencies to define legal partitions into pipeline stages. Given such a legal partitioning into pipeline stages, the presented synthesis procedure automatically identifies potential pipeline conflicts and implements code for forwarding (if possible) while stalling is implicitly given by the FIFO buffers. Finally, the sequential threads for the conflict-free pipeline stages are implemented in OpenMP-based C-code.

However, these researches have not considered time-predictable properties. The mapping from their multi-threaded code to multi-core platforms is handled by the underlying system. In addition, architectural aspects are not addressed, and consequently architectural based properties, e.g., time determinism, cannot be explicitly controlled in order to have precise and safe timing properties.

8 Conclusions and future work

This paper reports a SIGNAL compiler prototype based on the intermediate representation S-CGA. Since SIGNAL is polychronous, each variable can have its own clock. Moreover, the variables can be evaluated only at some instants which define their so-called clocks. In contrast to the SIGNAL language, clocked guarded actions can evaluate a variable even if its clock does not hold. As a consequence, we propose a variant of clocked guarded actions, namely S-CGA, which constrains variable accesses as done by SIGNAL. S-CGA has the same structure as clocked guarded actions, but they have different semantics. The front-end of the compiler, i.e., the translation from SIGNAL to S-CGA, is presented. The proof of semantics preservation mechanized in the theorem prover Coq is also given. Moreover, we present the back-end of the compiler, including sequential code generation and multi-threaded code generation. Concerned with the sequential code generation, we adapt the code generation to the S-CGA context. Meanwhile, we also consider enhancements of the compiler and their insertion in the compilation chain. Moreover, we propose an appropriate modular architecture for our prototype. One benefit of this approach is that we just need to redo a part of proof when some modules of the compilation process are changed. The widespread advent of multi-core processors further aggravates the complexity of timing analysis. This paper proposes the multi-threaded code generation by considering time-predictable properties. Our method integrates time predictability across several design layers, i.e., synchronous programming, verified compiler, and time-predictable multi-core architecture model.

Interaction among cores might also jeopardize software isolation layers, such as the one defined in ARINC653. There are some existing work, such as [42,44–46], about AADL modeling on multi-core architectures and their association with ARINC653.

Again, for us, the challenge will be to specify formally such a platform, with respect to space and time isolation, and to prove the satisfaction of timing properties at the application level.

Acknowledgements This work was partially supported by the National Natural Science Foundation of China (Grant No. 61502231), the Natural Science Foundation of Jiangsu Province (BK20150753), the National Key Basic Research Program of China (973 plan) (2014CB744904), Project of the State Key Laboratory of Software Development Environment of China (SKLSDE-2015KF-04), the Avionics Science Foundation of China (2015ZC52027), China Postdoctoral Science Foundation, the Collaborative Innovation Center of Novel Software Technology and Industrialization, and the RTRA STAE Foundation in France (http://www.fondation-stae.net/).

References

1. Potop-Butucaru D, de Simone R, Talpin J P. The synchronous hypothesis and synchronous languages. The Embedded Systems Handbook, 2005: 1–21
2. Boussinot F, de Simone R. The ESTEREL language. Proceedings of the IEEE, 1991, 79(9): 1293–1304
3. Halbwachs N, Caspi P, Raymond P, Pilaud D. The synchronous data flow programming language LUSTRE. Proceedings of the IEEE, 1991, 79(9): 1305–1320
4. Schneider K. The synchronous programming language QUARTZ. Internal Report 375. Kaiserslautern: University of Kaiserslautern, 2010
5. Benveniste A, Le Guernic P, Jacquemot C. Synchronous programming with events and relations: the SIGNAL language and its semantics. Science of Computer Programming, 1991, 16(2): 103–149
6. Dijkstra E W. Guarded commands, nondeterminacy and formal derivation of programs. Communications of the ACM, 1975, 18(8): 453–457
7. Brandt J, Gemünde M, Schneider K, Shukla S K, Talpin J P. Integrating system descriptions by clocked guarded actions. In: Proceedings of 2011 IEEE Forum on Specification and Design Languages. 2011, 1–8
8. Brandt J, Schneider K. Separate translation of synchronous programs to guarded actions. Technische Universität Kaiserslautern. Fachbereich Informatik, 2011
9. Brandt J, Schneider K, Shukla S K. Translating concurrent action oriented specifications to synchronous guarded actions. ACM SIGPLAN Notices, 2010, 45(4): 47–56
10. Edwards S, Tardieu O. SHIM: a deterministic model for heterogeneous embedded systems. IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 2006, 14(8): 854–867
11. Brandt J, Gemünde M, Schneider K, Shukla S K, Talpin J P. Representation of synchronous, asynchronous, and polychronous components by clocked guarded actions. Design Automation for Embedded Systems, 2012, 18(1–2): 63–97
12. SACRES consortium. The declarative code DC+, version 1.4. Esprit Project EP 20897: Sacres. 1997
13. Besnard L, Gautier T, Talpin J P. Code generation strategies in the Polychrony environment. Research Report RR-6894. 2009
14. Jose B A, Patel H D, Shukla S K, Talpin J P. Generating multi-threaded code from polychronous specifications. Electronic Notes in Theoretical Computer Science, 2009, 238(1): 57–69

15. Jose B, Shukla S K, Patel H D, Talpin J P. On the deterministic multi-threaded software synthesis from polychronous specifications. In: Proceedings of the 6th ACM & IEEE International Conference on Formal Methods and Models for Co-Design. 2008, 129–138
16. Papailiopoulou V, Potop-Butucaru D, Sorel Y, De Simone R, Besnard L, Talpin J P. From design-time concurrency to effective implementation parallelism: the multi-clock reactive case. In: Proceedings of Electronic System Level Synthesis Conference. 2011, 1–6
17. Hu K, Zhang T, Yang Z B. Multi-threaded code generation from Signal program to OpenMP. Frontiers of Computer Science, 2013,7(5): 617–626
18. SAE. AS5506A: Architecture Analysis and Design Language (AADL) Version 2.0. 2009
19. Leroy X. Mechanized semantics for compiler verification. Lecture Notes in Computer Science, 2012, 7679: 4–6
20. Pnueli A, Siegel M, Singerman E. Translation validation. In: Proceedings of the 4th International Conference on Tools and Algorithms for the Construction and Analysis of Systems. 1998, 151–166
21. Ngo V C, Talpin J P, Gautier T, Le Guernic P. Besnard L. Formal verification of synchronous data-flow program transformations toward certified compilers. Frontiers of Computer Science, 2013, 7(5): 598–616
22. Izerrouken N, Pantel M, Thirioux X. Machine-checked sequencer for critical embedded code generator. In: Proceedings of the 11th International Conference on Formal Methods and Software Engineering. 2009, 521–540
23. Besnard L, Gautier T, Le Guernic P. SIGNAL V4 Reference Manual. http: //www.irisa.fr/espresso/Polychrony/document/V4 def.pdf. 2010
24. Gamatié A. Designing Embedded Systems with the Signal Programming Language: Synchronous, Reactive Specification. Springer Science & Business Media. 2009
25. Le Guernic P, Gautier T. Data-flow to von Neumann: the Signal approach. Advanced Topics in Data-Flow Computing, 1991, 413–438,
26. Le Guernic P, Talpin J P, Le Lann J C. Polychrony for system design. Journal of Circuits, Systems, and Computers, 2003, 12(03): 261–303
27. Yang Z B, Bodeveix J P, Filali M. A comparative study of two formal semantics of the SIGNAL language. Frontiers of Computer Science, 2013, 7(5): 673–693
28. Yang Z B, Hu K, Ma D F, Bodeveix J P, Pi L, Talpin J P. From AADL to timed abstract state machines: a verified model transformation. Journal of Systems and Software, 2014, 93: 42–68
29. Yang Z B, Bodeveix J P, Filali M, Hu K, Ma D F. A verified transformation: from polychronous programs to a variant of clocked guarded actions. In: Proceedings of the 17th ACM International Workshop on Software and Compilers for Embedded Systems. 2014, 128–137
30. Feautrier P, Gamatié A, Gonnord L. Enhancing the compilation of synchronous dataflow programs with a combined numerical-boolean abstraction. CSI Journal of Computing, 2012, 1(4): 86–99
31. Gamatié A, Gautier T, Le Guernic P. Toward static analysis of SIGNAL programs using interval techniques. In: Proceedings of Synchronous Languages, Applications, and Programming. 2006.
32. Axer P, Ernst R, Falk H, Girault A, Grund D, Guan N, Jonsson B, Marwedel P, Reineke J, Rochange C, Sebastian M, Von Hanxleden R, Wilhelm R, Yi W. Building timing predictable embedded systems. ACM Transactions on Embedded Computing Systems, 2014, 13(4): 82
33. Wilhelm R, Engblom J, Ermedahl A, Holsti N, Thesing S, Whalley D, Bernat G, Ferdinand C, Heckmann R, Mitra T, Mueller F, Puaut I, Puschner P, Staschulat J, Stenström P. The worst-case execution-time problem-overview of methods and survey of tools. ACM Transactions on Embedded Computing Systems, 2008, 7(3): 36
34. Thiele L, Wilhelm R. Design for timing predictability. Real-Time Systems, 2004, 28(2–3): 157–177
35. Potop-Butucaru D, Caillaud B, Benveniste A. Concurrency in synchronous systems. Formal Methods in System Design, 2006, 28(2): 111–130
36. Besnard L, Gautier T, Le Guernic P, Talpin J P. Compilation of polychronous data flow equations. In: Shukla S K, Talpin J P, eds. Synthesis of Embedded Software. Springer US, 2010
37. Baudisch D, Brandt J, Schneider K. Dependency-driven distribution of synchronous programs. IFIP Advances in Information and Communication Technology, 2010, 329: 169–180
38. Baudisch D, Brandt J, Schneider K. Multithreaded code from synchronous programs: extracting independent threads for OpenMP. In: Proceedings of the Conference on Design, Automation and Test in Europe. 2010, 949–952
39. Baudisch D, Brandt J, Schneider K. Multithreaded code from synchronous programs: generating software pipelines for OpenMP. In: Proceedings of Workshop Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen (MBMV). 2010, 11–20
40. Schoeberl M, Huber B, Puffitsch W. Data cache organization for accurate timing analysis. Real-Time Systems, 2013, 49(1): 1–28
41. Schoeberl M. A time predictable instruction cache for a Java processor. Lecture Notes in Computer Science, 2004, 3292: 371–382
42. Delange J, Feiler P. Design and analysis of multi-core architecture for cyber-physical systems. In: Proceedings of the 7th European Congress Embedded Real Time Software and Systems (ERTSS). 2014.
43. Ngo V C, Talpin J P, Gautier T, Le Guernic P, Besnard L. Formal verification of compiler transformations on polychronous equations. Lecture Notes in Computer Science, 2012, 7321: 113–127
44. Hugues J. AADLib, a library of reusable AADL models. SAE Technical Paper, 2013
45. Gamatié A, Gautier T. Synchronous modeling of avionics applications using the SIGNAL language. In: Proceedings of the 9th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS). 2003, 144–151
46. Gamatié A, Gautier T, Guernic P L, Talpin J P. Polychronous design of embedded real-time applications. Transactions on Software Engineering and Methodology, 2007, 16(2): 9

Zhibin Yang is an assistant professor at Nanjing University of Aeronautics and Astronautics, China. He received his PhD degree in computer science from Beihang University, China in 2012. From 2012 to 2014, he was a Postdoc in IRIT of University of Toulouse, France. His research interests include safety-critical real-time system, formal verification, AADL, and synchronous languages.

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

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

Kai Hu is an associate professor in Beihang University (BUAA), China. He received his PhD from Beihang University in 2001. From 2001 to 2004, he did his post-doctoral research at Nanyang Technological University, Singapore. Since 2004, he is the leader of the team of LDMC in the Institute of Computer Architecture (ICA), BUAA. His research interests concern embedded real time systems and high performance computing.

Yongwang Zhao is an assistant professor at Beihang University (BUAA), China. He received his PhD degree in computer science from BUAA in 2009. His research interests include formal methods, real-time operating systems, and AADL.

Dianfu Ma is a professor at Beihang University, China. He was the executive director of Chinese Computer Federation, the secretary of the steering committee of Computer Science and Technology Education in Ministry of Education of China. He is the vice director of SOA standards working group under the steering committee of China National Information Technology Standardization. He took charge of the National Basic Research Program (also called 973 Program), National High-tech 863 Program, National Natural Science Foundation of China, Key Technologies Research and Development Program, etc. He has published more than 50 academic papers in international journals or conferences. He received the 3rd prize of Science and Technology Innovation Award from Ministry of Education of China in 2003, and 1st prize of Science and Technology Innovation Award of Beijing in 2011. His research interesting includes services computing, real-time systems, and high dependable software.