A Verified Transformation: From Polychronous Programs to a Variant of Clocked Guarded Actions

Zhibin Yang, Jean-Paul Bodeveix, M Filali, Kai Hu, Dianfu Ma

A Verified Transformation: from Polychronous Programs to a Variant of Clocked Guarded Actions

Zhibin Yang
IRIT, Université de Toulouse
Zhibin.Yang@irit.fr

Jean-Paul Bodeveix
IRIT, Université de Toulouse
bodeveix@irit.fr

Mamoun Filali
IRIT-CNRS, Université de Toulouse
filali@irit.fr

Kai Hu
School of Computer Science and Engineering, Beihang University, Beijing
hukai@buaa.edu.cn

Dianfu Ma
School of Computer Science and Engineering, Beihang University, Beijing
dfma@buaa.edu.cn

ABSTRACT

SIGNAL belongs to the synchronous languages family. Such languages 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 key step of a verified SIGNAL compiler prototype, that is the transformation from a subset of SIGNAL to S-CGA (a variant of clocked guarded actions) and the proof of semantics preservation. Compared with the existing SIGNAL compiler, we use clocked guarded actions as the intermediate representation, to integrate more synchronous programs into our verified 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. Thus, we propose a variant of clocked guarded actions, namely S-CGA, which constrains variable accesses as done by SIGNAL. To conform with the revised semantics of clocked guarded actions, we also do some adjustments on the existing translation rules from SIGNAL to clocked guarded actions. Finally, the verified transformation is mechanized in the theorem prover Coq.

Keywords

Synchronous languages, Verified compiler, SIGNAL, Guarded actions, Semantics preservation

1. INTRODUCTION

Safety-critical real-time systems such as avionics, space systems, and nuclear power plants, are also 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 arrival time of events may be different, and the computation needs time. The synchronous approach is an important choice for the design of these systems, which relies on the *synchronous hypothesis* [25]. Firstly, the computation time is abstracted as zero, that lets system behaviors be divided into a discrete sequence of instants. At each instant, the system does input-computation-output, which takes zero time. Secondly, the different arrival time of events are abstracted as the relative order between events. 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.

There are several synchronous languages, such as ESTEREL [3], LUSTRE [16] and QUARTZ [27] based on the *perfect synchrony* paradigm, and SIGNAL [1] based on the *polychrony* paradigm. Halfway between LUSTRE and SIGNAL, we can cite Lucid Synchrone [26] which allows clock synthesis and oversampling. These languages can be considered as different implementations of the synchronous hypothesis.

However, an integration infrastructure for different synchronous languages has gained a lot of interests in recent years [4][5][28]. It is not only possible to deepen the understanding of these different models of computation but more importantly, one can reuse a common verification, simulation and synthesis back-end tool for different models. A classical solution used e.g. in most compilers is to use a common intermediate representation, which bridges the gap between powerful programming languages with complex semantics and the low-level description of the target code.

Guarded commands [9], called *asynchronous guarded actions* by J. Brandt et al. [4], are a well-established concept for the description of concurrent systems. In the spirit of the guarded commands, J. Brandt et al. propose *synchronous guarded actions* [7] 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 [8] or SHIM [10]), they propose an extended intermediate representation, that is *clocked guarded actions* [4][5] 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. [6] presents an embedding of polychronous programs into synchronous ones. The embedding gives us access to the methods and tools that already exist for synchronous specifications, especially for simulation and verification (as provided by the Averest system[1]). Moreover, we can also do a translation in another direction, that is the translation of synchronous programs into polychronous ones. It can be considered as a desynchronization of synchronous programs to GALS architectures through reusing the existing Polychrony[2] tool.

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[3] KCG automatic C code generator has been qualified as a development tool at DO-178B level A. Our work mainly focuses on the SIGNAL language. Following the global idea, 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 [18]. The intermediate representation is a variant of clocked guarded actions (called S-CGA), and currently the target is sequential C code (multi-threaded code will be considered in the future). This paper reports one of the transformations of the verified SIGNAL compiler prototype, that is the transformation from the subset of SIGNAL to S-CGA and the proof of semantics preservation. There exist several semantics for SIGNAL, such as denotational semantics based on traces (called trace semantics) [2][12][14], denotational semantics based on tags which puts forward a partial order view of time (called tagged model semantics) [12][15], structural operational semantics defining inductively a set of possible transitions [1][12], operational semantics defined by synchronous transition systems (STS) [23]. [31] study the equivalence between the trace semantics and the tagged model semantics, to assert a determined and precise semantics of the SIGNAL language. [28] defines a unified constructive semantic framework to unite QUARTZ and SIGNAL. This framework allows us to better understand the relationship between synchrony and polychrony. Along the way, they define an executable operational semantics of SIGNAL. The relation between our work and related work is shown in Figure 1 (which extends the Figure given in [4]).

The contribution of this paper is the study of a verified SIGNAL compiler prototype, while existing formal verification techniques around SIGNAL are mainly based on translation validation [23][22]. Furthermore, compared with the existing SIGNAL compiler-Polychrony, we use clocked guarded actions as the intermediate representation, to integrate more synchronous programs such as QUARTZ, AIF[4] [4] into our verified 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. Thus, we propose a variant of clocked guarded actions, namely S-CGA, which constrains variable accesses as done by SIGNAL. 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 [4][5]). Moreover, the verified transformation is mechanized in the theorem prover Coq.

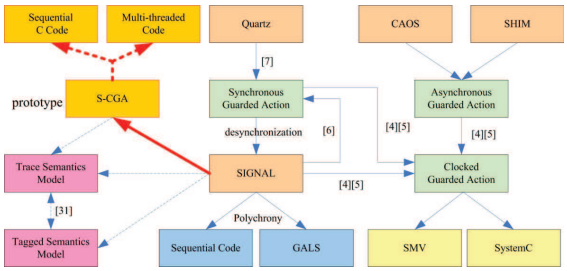

Figure 1: A global view of the relation between our work and related work

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 translation rules. The proof of the semantics equivalence of the transformation is presented in Section 5. Section 6 gives an overview of our compiler prototype. 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 As declared 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. So, 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 $\perp$). 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, etc.

[1]http://www.averest.org
[2]http://www.irisa.fr/espresso/Polychrony
[3]http://www.esterel-technologies.com/

[4]Averest Intermediate Format

Example 1 Three signals named $input_1$, $input_2$, $output$ are shown as follows.

$input_1$	1	$\perp$	3	$\perp$	$\cdots$
$input_2$	$\perp$	5	7	9	$\cdots$
$output$	$\perp$	$\perp$	10	$\perp$	$\cdots$

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 clock 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.

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).

- Monoclock operators, including *instantaneous function* and *delay*. The instantaneous function $x := f(x_1, \cdots, x_n)$ applied on a set of inputs $x_1, \cdots, x_n$ will produce the output x, while the delay operator $x := x_1 \$ init\ c$ sends a previous value of the input to the output with an initial value c.

- Multiclock operators, including *undersampling* and *deterministic merging*. The undersampling operator $x := x_1\ when\ x_2$ is used to check 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.

- Clock synchronization, the equation $x_1 \hat{}= x_2 \hat{}= \cdots \hat{}= x_n$ specifies that signals $x_1, x_2, \cdots, x_n$ are synchronous.

- 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.

- 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, so we just consider the primitive constructs, that is kernel SIGNAL (kSIGNAL for short). Its abstract syntax is presented as follows.

$P ::=$	$x := f(x_1, \cdots, x_n)$	(instantaneous function)
	$\|x := x_1 \$ init\ c$	(delay)
	$\|x := x_1\ when\ x_2$	(undersampling)
	$\|x := x_1\ default\ x_2$	(deterministic merging)
	$\|P\|P'$	(composition)

We should notice that we can use both primitive constructs and extended constructs in the subset of SIGNAL. However, the compiler will translate it into kSIGNAL (just use primitive constructs). So, in the proof of semantics preservation, we consider kSIGNAL and S-CGA.

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, etc. This paper considers the trace semantics. In the following paragraphs, we summarize the trace model [2][12] 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 object 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.

After, the definition of traces is given. 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 ($X \to \{\bot\}$), where all the signals are absent at a logical instant, is denoted $\bot_e(X)$. Moreover, the set of *events* on X ($X \to \mathcal{V}_X^{\bot}$) 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 ($\mathbb{N} \to \{\bot_e(X)\}$), i.e., the infinite sequence formed by the infinite repetition of $\bot_e(X)$, is denoted $\bot_X$.

Definition 4 (Process) Given a SIGNAL process, its trace semantics, denoted *Process*, 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 $x := f(x_1, \cdots, x_n)$ is defined as follows:

$$\forall i \in \mathbb{N}$$
$$x_i = \begin{cases} \bot & if\ x_{1i} = \ldots = x_{ni} = \bot \\ f(x_{1i}, \ldots, x_{ni}) & if\ x_{1i} \neq \bot \wedge \ldots \wedge x_{ni} \neq \bot \end{cases}$$

At each instant i, 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_i gets the value of $f(x_{1i}, \ldots, x_{ni})$ when the signals are all present. The function f includes different mathematical operations, such as arithmetic operations, boolean operations, etc.

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

$$\begin{array}{l} - \ (\forall i \in \mathbb{N})\ x_{1i} = \bot \Leftrightarrow x_i = \bot \\ - \ \{k \mid x_{1k} \neq \bot\} \neq \emptyset \Rightarrow x_{\min\{k \mid x_{1k} \neq \bot\}} = c \\ - \ (\forall i \in \mathbb{N})\ x_{1i} \neq \bot \wedge \{k > i \mid x_{1k} \neq \bot\} \neq \emptyset \\ \quad \Rightarrow x_{\min\{k > i \mid x_{1k} \neq \bot\}} = x_{1i} \end{array}$$

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 i, x takes the most recent value of x_1 except the one at i. 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:

$$\forall i \in \mathbb{N}$$
$$x_i = \begin{cases} x_{1i} & if\ x_{2i} = true \\ \bot & otherwise \end{cases}$$

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 $x = x_1 \hat{}* [x_2]$, 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:

$$\forall i \in \mathbb{N}$$
$$x_i = \begin{cases} x_{1i} & if\ x_{1i} \neq \bot \\ x_{2i} & otherwise \end{cases}$$

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 i, x_i gets the merge of the values of x_{1i} and x_{2i}, and the value of x_{1i} has a higher priority.

Finally, we apply these semantics rules to a SIGNAL process, to get a complete semantics of the process, that is *Process* (*Definition* 4).

3. S-CGA

In the papers [4] and [5], clocked guarded actions has been defined as a common representation for synchronous (via synchronous guarded actions), polychronous and asynchronous (via asynchronous guarded actions) models. It has multi-clocked feature. However, in contrast to the SIGNAL language, clocked guarded actions can evaluate a variable even if its clock does not hold. As a consequence, we introduce S-CGA, which is a variant of clocked guarded actions. S-CGA constrains variable accesses as done by SIGNAL. 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.

3.1 Abstract Syntax of S-CGA

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

Definition 6 (S-CGA) A 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:

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

where

- the guard γ is a Boolean condition over the variables of X, their respective clocks (for a variable $x \in X$, we denote its clock $\hat{x}$), and their respective initial clocks (denoted $init(\hat{x})$),
- 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[5] and its value is true, then x is present and τ is defined. Moreover, $init(\hat{x})$ exactly holds the first instant when x is present.

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 $\|$.

Remark 1 The $\Rightarrow$ is S-CGA syntactic token for separating the guard and the action, later we use $\rightarrow$ for the logical implication.

To define the semantics of S-CGA precisely, we first present the syntax of guards (γ), assumptions (σ) and expressions (τ) in detail.

Definition 7 The syntax of γ, σ, and τ are as follows.

$$\gamma, \sigma ::= init(\hat{x}) \mid x \mid \hat{x} \mid f(\gamma, ..., \gamma)$$
$$\tau ::= x \mid f(\tau, ..., \tau)$$

Note that, in this definition, f can represent a constant ($true$, $false$, ...), an arithmetic or logical operator, and more generally an user-defined function.

3.2 Denotational Semantics of S-CGA

Firstly, we introduce the clock of the terms such as guards (γ), assumptions (σ) and expressions (τ). We consider here that guards and assumptions have the same syntax.

Definition 8 Given a trace S and an instant i, we define the following functions:

- $\widehat{[\![\gamma]\!]}_{S,i}$ defines the domain of $[\![\gamma]\!]_{S,i}$. $\widehat{[\![\gamma]\!]}_{S,i}$ is true in S at instant i if all the variables of γ are present at that instant. Note that $\widehat{[\![\hat{x}]\!]}_{S,i}$ is defined to be true because a clock can be read at any instant.

$$\widehat{[\![\gamma]\!]}_{S,i} : \begin{cases} \widehat{[\![init(\hat{x})]\!]}_{S,i} = true \\ \widehat{[\![f(\gamma_1, ..., \gamma_n)]\!]}_{S,i} = \widehat{[\![\gamma_1]\!]}_{S,i} \wedge ... \wedge \widehat{[\![\gamma_n]\!]}_{S,i} \\ \widehat{[\![x]\!]}_{S,i} = (S(i,x) \neq \perp) \\ \widehat{[\![\hat{x}]\!]}_{S,i} = true \end{cases}$$

- $[\![\gamma]\!]_{S,i}$ is a partial function defined when $\widehat{[\![\gamma]\!]}_{S,i}$, and it is used to compute the value of γ (true or false).

$$[\![\gamma]\!]_{S,i} : \begin{cases} [\![init(\hat{x})]\!]_{S,i} = S(i,x) \neq \perp \\ \qquad\qquad \wedge \forall j < i, S(j,x) = \perp \\ [\![f(\gamma_1, ..., \gamma_n)]\!]_{S,i} = f([\![\gamma_1]\!]_{S,i}, ..., [\![\gamma_n]\!]_{S,i}) \\ [\![x]\!]_{S,i} = S(i,x) \\ [\![\hat{x}]\!]_{S,i} = S(i,x) \neq \perp \end{cases}$$

[5]An expression is said to be defined if all the variables it contains are present.

- $\widehat{[\![\tau]\!]}_{S,i}$ defines the domain of $[\![\tau]\!]_{S,i}$. $\widehat{[\![\tau]\!]}_{S,i}$ is true in S at instant i if all the variables of τ are present at that instant.

$$\widehat{[\![\tau]\!]}_{S,i} : \begin{cases} \widehat{[\![f(\tau_1, ..., \tau_n)]\!]}_{S,i} = \widehat{[\![\tau_1]\!]}_{S,i} \wedge ... \wedge \widehat{[\![\tau_n]\!]}_{S,i} \\ \widehat{[\![x]\!]}_{S,i} = (S(i,x) \neq \perp) \end{cases}$$

- $[\![\tau]\!]_{S,i}$ is a partial function defined when $\widehat{[\![\tau]\!]}_{S,i}$, and it is used to compute the value of τ.

$$[\![\tau]\!]_{S,i} : \begin{cases} [\![f(\tau_1, ..., \tau_n)]\!]_{S,i} = f([\![\tau_1]\!]_{S,i}, ..., [\![\tau_n]\!]_{S,i}) \\ [\![x]\!]_{S,i} = S(i,x) \end{cases}$$

Secondly, based on the trace model (section 2.2) and the definitions given as above, we give the trace semantics of S-CGA.

Definition 9 (Trace semantics of S-CGA) The trace semantics of a 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,

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

(2) $[\![\gamma \Rightarrow next(x) = \tau]\!]_S =$
$\forall i_1 < i_2 \in \mathbb{N},$
$((\forall i' \in \mathbb{N},\ i_1 < i' < i_2 \rightarrow \neg \widehat{[\![x]\!]}_{S,i'}) \wedge \widehat{[\![\gamma]\!]}_{S,i_1} \wedge [\![\gamma]\!]_{S,i_1})$
$\rightarrow (\widehat{[\![x]\!]}_{S,i_1} \wedge \widehat{[\![\tau]\!]}_{S,i_1} \wedge (\widehat{[\![x]\!]}_{S,i_2} \rightarrow [\![x]\!]_{S,i_2} = [\![\tau]\!]_{S,i_1}))$

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

- 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 i_1, x and τ are present at i_1, and if i_2 is the next instant where x is present, then the value of x at i_2 is that of τ at instant i_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 $[\![scga_1 \parallel scga_2]\!]_S = [\![scga_1]\!]_S \wedge [\![scga_2]\!]_S$.

Remark 2 In S-CGA, we check if a variable is present or absent. For a *guard* γ, we compute its value when it is present. However, clocked guarded actions can evaluate a variable even if its clock does not hold. In this case, its value is the one of the last instant it was defined.

- For the *immediate assignment*, we do the assignment $x = \tau$ when x and τ are present, else we don't do it. However, clocked guarded actions do the assignment even if the clock of τ does not hold.

- For the *delay assignment*, we should notice that the original semantics given in [5] leads to non-natural constraints on the model. Actually, when writing $\gamma \Rightarrow next(x) = \tau$, should γ becomes twice or more satisfied before the next instant of x raises an inconsistency if τ has different values (See Example 2). Thus, in S-CGA, we have the constraints $\gamma \rightarrow \hat{x}$ and $\gamma \rightarrow \hat{\tau}$.
- For the *assumption*, we have the constraint $\gamma \rightarrow \hat{\sigma}$.

Example 2 An example based on the original semantics of $\gamma \Rightarrow next(x) = \tau$.

	i_1	i'	$\cdots$	i_2	$\cdots$
τ	v_1	v_2	$\cdots$		$\cdots$
x	v	$\perp$	$\perp$	$x=?$	$\cdots$
γ	$true$	$true$	$\cdots$		$\cdots$

At the instant i_1, the guard γ is satisfied, τ is evaluated and its value is v_1. The next instant of $\hat{x}$ is i_2. However, γ is also satisfied in the instants during i_1 and i_2. For instance, at the instant i', γ is satisfied and the value of τ is v_2. It implies that v_1 should be equal to v_2.

4. TRANSLATION FROM KSIGNAL TO S-CGA

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 [4]. 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) $x := f(x_1, \cdots, x_n)$	$\Rrightarrow$	$\hat{x} \Rightarrow x = f(x_1, \cdots, x_n)$ $\| \widehat{x_1} \Rightarrow assume(\hat{x})$ $\| \ ...$ $\| \widehat{x_n} \Rightarrow assume(\hat{x})$
(2) $x := x_1 \ \$ \ init \ c$	$\Rrightarrow$	$init(\hat{x}) \Rightarrow x = c$ $\| \ \hat{x} \Rightarrow next(x) = x_1$ $\| \ true \Rightarrow assume(\hat{x} = \widehat{x_1})$
(3) $x := x_1 \ when \ x_2$	$\Rrightarrow$	$\widehat{x_1} \wedge x_2 \Rightarrow x = x_1$ $\| \ \hat{x} \Rightarrow assume(\widehat{x_1} \wedge x_2)$
(4) $x := x_1 \ default \ x_2$	$\Rrightarrow$	$\widehat{x_1} \Rightarrow x = x_1$ $\| \ \widehat{x_2} \wedge \neg\widehat{x_1} \Rightarrow x = x_2$ $\| \ \hat{x} \Rightarrow assume(\widehat{x_1} \vee \widehat{x_2})$

- Translation (1): The *instantaneous function* is applied to the inputs and produces the output. Note that the immediate assignment $\hat{x} \Rightarrow x = f(x_1, \cdots, x_n)$ implicitly imposes $\hat{x} \Rightarrow \widehat{x_1}, ..., \hat{x} \Rightarrow \widehat{x_n}$, so in the assumption we only assert $\widehat{x_1} \Rightarrow assume(\hat{x}), ..., \widehat{x_n} \Rightarrow assume(\hat{x})$. 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* 9), $assume(\widehat{x_1} \wedge x_2)$ 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, 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 3 Compared with the translation rules given in [4]. The main change is in the Translation (3), $true \Rightarrow assume(\hat{x} = \widehat{x_1} \wedge \widehat{x_2} \wedge x_2)$ has been changed into $\hat{x} \Rightarrow assume(\widehat{x_1} \wedge x_2)$. According to the Rule (3) (*Definition* 9): 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 and its value is true, $\widehat{x_1} \wedge x_2$ is present and true.

5. THE PROOF OF SEMANTICS PRESERVATION

We envision the extraction of a complete verified-compiler prototype from the mechanization in the future. Thus, we would like to use the theorem prover Coq, to express and verify the transformation from kSIGNAL to S-CGA. In this section, we first give a brief introduction to Coq. Then, we give the proof of semantics preservation of the transformation.

5.1 A Brief Introduction to Coq

Coq [30] is a theorem prover based on the Calculus of Inductive Constructions (CiC) which is a variant of type theory, following the "Curry-Howard Isomorphism" paradigm, enriched with support for inductive and co-inductive definitions of data types and predicates. From the specification perspective, Coq offers a rich specification language to define problems and state theorems. From the proof perspective, proofs are developed interactively using tactics, which can reduce the workload of the users. Moreover, the type-checking performed by Coq is the key point of proof verification.

Here, we try to give an intuitive introduction to the Coq terminologies which are used in this paper. In the spirit of "Curry-Howard Isomorphism" paradigm, types may represent programming data-types or logical propositions. So, the Coq objects used in this paper can be sorted into two categories: the *Type* sort and the *Prop* sort:

- *Type* is the sort for data types and mathematical structures, i.e. well-formed types or structures are of type *Type*. Data types can be basic types such as *nat*, *bool*, $nat \rightarrow nat$, etc., and can be *inductive* structures, *record* and *co-inductive* structures (for infinite objects, as for example infinite traces). We use *Fixpoint* and *CoFixpoint* definitions to define functions over inductive and to co-inductive data types.

- *Prop* is the sort for propositions, i.e. well-formed propositions are of type *Prop*. We can define new predicates using *inductive*, *record* (for conjunctions of properties) or *co-inductive* definitions.

5.2 Semantics Preservation

As shown in Figure 2, the Coq mechanization includes 7 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.

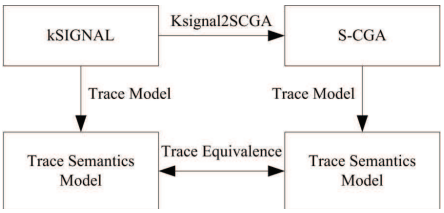

Figure 2: The global view of the semantics preservation

All the definitions given in this paper 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).

For the *instantaneous function* construct. (1) Its trace semantics is defined as *Sassignment*. (2) As defined in section 4, its translated guarded actions are $\hat{x} \Rightarrow x = f(x_1, \cdots, x_n)$, $\widehat{x_1} \Rightarrow assume(\hat{x})$, ..., and $\widehat{x_n} \Rightarrow assume(\hat{x})$. Applying the semantics of S-CGA (*scgaSimm* is the semantics rule of immediate assignment), we can get the semantics of instantaneous function construct by translating into S-CGA. Then, we prove the trace equivalence between (1) and (2).

```
Lemma 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 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.
```

For the *delay* construct. (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 rule of delayed assignment, and *scgaSctr* is the semantics rule of assumption), we can get the semantics of delay construct by translating 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 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)))
 ∧ scgaSctr gTrue (^x ^= ^x1) tr (getFirst0 tr).

Lemma 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.
```

For the *undersampling* construct. (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 by translating 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 signal2scga_when1: ∀ x x1 x2 tr,
 Swhen x x1 x2 tr →
 scgaSimm (^x1 ^* x2) x x1 tr
 ∧ scgaSctr ^x (^x1 ^* x2) tr.

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

For the *deterministic merging* construct. (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 by translating 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 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 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*, that the two semantics models, i.e., *(Process2Sprocess P)* and *(scga2Sprocess(signal2scga P))* are trace equivalent (they have 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 (p1 p2:Sprocess):Prop:=
{
```

```
  SPd: ∀ y, sdom p1 y ↔ sdom p2 y;
  SPs: ∀ tr, straces p1 tr ↔ straces p2 tr
}.

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

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

6. TOWARDS A VERIFIED COMPILER PROTOTYPE

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 [15], such as the synchronizations of each process, i.e., an equation system of relations 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 [11], the authors have shown that there is a limitation of the clock calculus of the SIGNAL compiler. For example, for the under-sampling 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 or defined by a complex boolean function, $[x_2]$ 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, that is *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 [13], 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 Figure 3, in our compiler prototype: (1) We have considered the main concepts of the clock calculus, such as the synchronizations of each process, resolution of a system of clock equations, and construction of a clock hierarchy. (2) To integrate more synchronous programs into our prototype in the future, we introduce S-CGA as the intermediate representation, and we rewrite the clock calculus on 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 enhancements such as [11][13], so in the resolution of a system of clock equations, there are two options *Cpt_rnf_on_BDD* and *Cpt_rnf_on_SMT*.

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

- Module 1: As mentioned in section 2.1,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: As a difference with 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 or a complex boolean function. 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, and it 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 in the SIGNAL compiler. Furthermore, we do some optimizations based on clock inclusions. For instance, there are two equations in a given process, $y = x$ *when* x_1 and $z = x$ *when* (x_1 *and* x_2), there is a clock-inclusion relation: $[x_1 \wedge x_2] \rightarrow [x_1]$, that's 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.

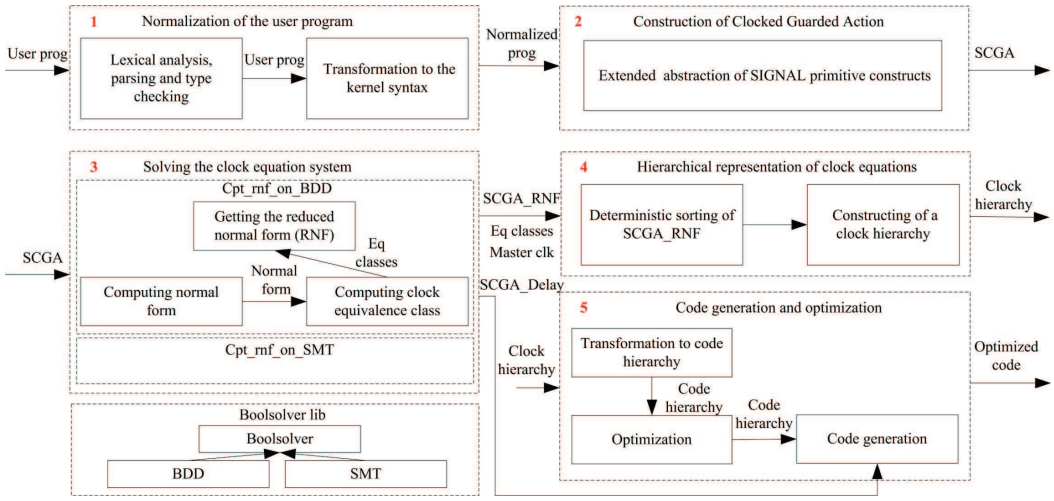

Figure 3: The architecture of the verified compiler prototype

7. RELATED WORK

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 for example 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 [20], by using translation validation [23], etc.

A. Pnueli et al. propose the notion of translation validation to verify the code generator of SIGNAL [23]. 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 [22], 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 the work of [21], 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. So it yields that one need to redo the validation when the source program is changed. Following the global idea, 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 [20].

8. CONCLUSION AND FUTURE WORK

This paper reports a key step of a verified SIGNAL compiler prototype, that is the transformation from a subset of SIGNAL to the intermediate representation S-CGA and the proof of semantics preservation. 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. Moreover, to conform with the revised semantics of clocked guarded actions, we also do some adjustments on the existing translation rules from SIGNAL to clocked guarded actions. Finally, the verified transformation is mechanized in the theorem prover Coq.

In the next step, we will rewrite in Coq the remaining modules of our compiler prototype, that is from S-CGA to sequential code. Moreover, in order to exploit the emerging multi-core processors, thanks to the theory of *weakly endochronous systems* [24], there are several research to synthesize multi-threaded code from the synchronous specifications [17][19][29]. However, one also needs to prove the semantics preservation from the SIGNAL specifications to

the multi-threaded code.

9. ACKNOWLEDGMENTS

This work was supported in part by the RTRA STAE Foundation in France (http://www.fondation-stae.net/).

We would like to thank Prof.Jean-Pierre Talpin, Prof.Paul Le Guernic, Prof.Loïc Besnard and Prof.Thierry Gautier for the fruitful discussions we have had.

10. REFERENCES

[1] A. Benveniste, P. L. Guernic, and C. Jacquemot. Synchronous programming with events and relations: the signal language and its semantics. *Science of Computer Programming*, 16:103–149, 1991.

[2] L. Besnard, T. Gautier, and P. L. Guernic. *SIGNAL V4 Reference Manual*, 2010.

[3] F. Boussinot and R. de Simone. The esterel language. *Proceedings of the IEEE*, 79(9):1293–1304, 1991.

[4] J. Brandt, M. Gemunde, K. Schneider, S. K. Shukla, and J.-P. Talpin. Integrating system descriptions by clocked guarded actions. In *FDL*, pages 1–8. IEEE, 2011.

[5] J. Brandt, M. Gemünde, K. Schneider, S. K. Shukla, and J.-P. Talpin. Representation of synchronous, asynchronous, and polychronous components by clocked guarded actions. *Design Automation for Embedded Systems*, pages 1–35, 2012.

[6] J. Brandt, M. Gemunde, K. Schneider, S. K. Shukla, and J.-P. Talpin. Embedding polychrony into synchrony. *IEEE Trans. Software Eng.*, 39(7):917–929, 2013.

[7] J. Brandt and K. Schneider. Separate translation of synchronous programs to guarded actions. *Internal Report 382/11, Department of Computer Science, University of Kaiserslautern*, 2011.

[8] J. Brandt, K. Schneider, and S. K. Shukla. Translating concurrent action oriented specifications to synchronous guarded actions. In *LCTES*, pages 47–56. ACM, 2010.

[9] E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457, 1975.

[10] S. A. Edwards and O. Tardieu. Shim: a deterministic model for heterogeneous embedded systems. *IEEE Trans. VLSI Syst.*, 14(8):854–867, 2006.

[11] P. Feautrier, A. Gamatié, and L. Gonnord. Enhancing the compilation of synchronous dataflow programs with a combined numerical-boolean abstraction. *CSI Journal of Computing*, 1(4):8:86 – 8:99, 2012.

[12] A. Gamatié. *Designing embedded systems with the SIGNAL programming language*. Springer, 2010.

[13] A. Gamatié, T. Gautier, and P. L. Guernic. Towards static analysis of signal programs using interval techniques. In *Synchronous Languages, Applications, and Programming, SLAP'06, Vienna - Austria*, 2006.

[14] P. L. Guernic and T. Gautier. Data-flow to von neumann: the signal approach. *Advanced Topics in Data-Flow Computing*, pages 413–438, 1991.

[15] P. L. Guernic, J.-P. Talpin, and J. christophe Le Lann. Polychrony for system design. *Journal of Circuits Systems and Computers*, 12:261–304, 2002.

[16] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data-flow programming language lustre. *Proceedings of the IEEE*, 79(9):1305–1320, 1991.

[17] K. Hu, T. Zhang, and Z. Yang. Multi-threaded code generation from Signal program to OpenMP. *Frontiers of Computer Science*, 7(5):617–626, 2013.

[18] N. Izerrouken, M. Pantel, and X. Thirioux. Machine-checked sequencer for critical embedded code generator. In *Formal Methods and Software Engineering*, volume 5885 of *Lecture Notes in Computer Science*, pages 521–540. Springer Berlin Heidelberg, 2009.

[19] B. A. Jose. *Formal model driven software synthesis for embedded systems*. PhD thesis, Virginia Polytechnic Institute and State Univeristy, 2011.

[20] X. Leroy. Mechanized Semantics for Compiler Verification. In *APLAS*, volume 7705 of *Lecture Notes in Computer Science*, pages 386–388. Springer, 2012.

[21] V. Ngo, J.-P. Talpin, T. Gautier, P. Guernic, and L. Besnard. Formal verification of compiler transformations on polychronous equations. In *Integrated Formal Methods*, volume 7321 of *Lecture Notes in Computer Science*, pages 113–127. Springer Berlin Heidelberg, 2012.

[22] V. Ngo, J.-P. Talpin, T. Gautier, P. Guernic, and L. Besnard. Formal verification of synchronous data-flow program transformations toward certified compilers. *Frontiers of Computer Science*, pages 1–19, 2013.

[23] A. Pnueli, M. Siegel, and F. Singerman. Translation validation. pages 151–166. Springer, 1998.

[24] D. Potop-Butucaru, B. Caillaud, and A. Benveniste. Concurrency in synchronous systems. *Formal Methods in System Design*, 28(2):111–130, 2006.

[25] D. Potop-Butucaru, R. de Simone, and J.-P. Talpin. The synchronous hypothesis and synchronous languages. *The Embedded Systems Handbook*, pages 1–21, 2005.

[26] M. Pouzet. *Lucid Synchrone, version 3. Tutorial and reference manual*. Université Paris-Sud, LRI, April 2006.

[27] K. Schneider. The synchronous programming language quartz. *Internal report, Department of Computer Science, University of Kaiserslautern, Germany*, 2010.

[28] J.-P. Talpin, J. Brandt, M. Gemünde, K. Schneider, and S. Shukla. Constructive polychronous systems. In *Logical Foundations of Computer Science*, volume 7734 of *Lecture Notes in Computer Science*, pages 335–349. Springer Berlin Heidelberg, 2013.

[29] V.Papailiopoulou, D.Potop-Butucaru, Y.Sorel, R. Simone, L.Besnard, and J.-P.Talpin. From design-time concurrency to effective implementation parallelism: the multi-clock reactive case. *Electronic System Level Synthesis Conference*, pages 1–6, 2011.

[30] B. Y. and C. P. *Interactive Theorem Proving and Program Development: Coq Art: The Calculus of Inductive Constructions*. Springer, 2004.

[31] Z. Yang, J.-P. Bodeveix, and M. Filali. A comparative study of two formal semantics of the signal language. *Frontiers of Computer Science*, 7(5):673–693, 2013.