Available online at www.sciencedirect.com

SciVerse ScienceDirect

# Verification of Concurrent Code from Synchronous Specifications

KAI HU[a], TENG ZHANG[ab], YI DING[c]*, JIAN ZHU[a], JEAN-PIERRE TALPIN[d]

[a] *State Key Laboratory of Software Development Environment, Beihang University, Beijing, China*
[b]*Univerisity of Pennsylvania, USA*
[c]*School of Information, Beijing Wuzi University, Beijing,China*
[d]*INRIA-Rennes, Campus de Beaulieu, Rennes, France*

**Abstract**

The synchronous language SIGNAL is a formal specification formalism for developing safety-critical real-time systems. It is a multi-clocked data-flow modeling language suitable for specifying deterministic concurrent behaviors. Its model of computation and communication very well matches recent trends to utilize multi-core processors for executing real-time systems, by taking advantage of its concurrent semantics. The SIGNAL compiler generates code from data-flow specifications while analyzing and verifying safety properties of the system under design: deadlock-freedom, determinism. However, most of recent works have focused on generating sequential code from SIGNAL. Choosing the parallel library OpenMP as the target, this paper proposes a methodology to generate and verify concurrent code automatically from SIGNAL specifications. This is done by first exploring clock relations among signals by application of a so-called clock calculus. Then, specifications are translated into EDGs (Equation-Dependency Graphs) to analyze global data-dependency relations. An EDG is then partitioned into concurrent tasks to help explore parallelism in the original specification while preserving its semantic. Combined with clock relations, parallel tasks are finally mapped onto the OpenMP structures. The proposed approach is illustrated by a realistic case study.
*Keywords:*Synchronous specifications, SIGNAL, parallel programming, OpenMP, code generation;

## 1. Introduction

Safety-critical real-time systems [1] in avionics, aerospace, and automotive control, are an area of research where formal methods, based on rigorous mathematical models, are often used to help engineers during the design of complex, distributed, real-time systems. Such systems are growingly modeled using, abstract, formal specification languages to allow safety-critical properties to be unambiguously stated and validated (by model

* Corresponding author.
*E-mail address:*dingyi@bwu.edu.cn

checking tools or semi-automated theorem proving) before to automatically generate code for the system specification. Model-based verification allows detecting design issues as early as possible and verified models allow automatically generating executable code to facilitate the engineer's job. Safety-critical real-time systems increasingly use on multi-core architectures, that offer tremendous performances at reasonable costs, yet with increasing functional and non-functional constraints for automated code generation. Many parallel programming models including MPI [2], OpenMP [3] and TBB [4] have been proposed and used to support execution of parallel applications on multi-core architecture. However, automated generation of parallel code from a formal specification is still a vast research area of open problems because of the heterogeneity of multi-core architectures and the complexity of parallelizing, placing, routing, scheduling code on such architectures.

Synchronous specification languages such as Esterel [7], Lustre [8], SIGNAL [9], QUARTZ [10], adopt the synchronous hypothesis. They provide an abstraction where consecutive, periodic, hardware time samples are represented by sequences of discrete events (or logically related instant). The model of the system in such a specification language is an infinite sensing-computing-actuating loop that interacts with the external environment continuously as a computer-controlled system. The synchronous hypothesis assumes that each sample of this sensing-computing-actuating loop (called reactions) can be completed within a bounded amount of time (the worst-case reaction time) and hence be abstracted as logical moment (logical relations between the input, test and output events). By using such an abstracted specification, one can describe the functional behavior of the system without considering its concrete execution platform, in order to model-check functional safety properties and generate safety-checked code.

As the only synchronous language with a multi-clocked semantics, SIGNAL expresses combinations of data-flow equations that are partially related in time. It is thus of an appealing model of concurrency to specify timely decentralized computation in a distributed or concurrent system. SIGNAL compiler Polychrony [11] allows to generate code from synchronous specifications, and then to verify that the generated code satisfies the specification, by using SAT-SMT-supported translation validation. However, the existing SIGNAL compiler does not yet support the verification of its distributed code generation functionalities. This paper proposes the automatic parallel code generation from SIGNAL specifications to a multi-core, parallel, OpenMP platform. Our main contributions are:

- A clock calculus method based on the analysis of Boolean equations is proposed. It takes data flow equations as input and then goes through the translation of data flow equations to clock equations, the analysis of clock equation sets, the generation of equivalence classes, and the generation of normative equations. When inserting a clock node, this paper uses implication detection technology to insert it into a deeper position in the tree to generate more efficient code.
- A parallel program generation method based on EDG is proposed. First, the definition of EDG and the method of generating EDG from the Signal data flow equation are given. Then we propose the method of parallel task division based on a topological sorting algorithm according to the characteristics of EDG, and the correct insertion of clock information into the task. It ensures that the signal runs under the correct clock schedule.
- We take a typical Signal program containing four core syntaxes as an example to show the process of generating parallel simulation code from the Signal program. Through code analysis, the effectiveness of the translation method is demonstrated. Then we did a performance comparison experiment to prove the high performance of the parallel code under the complex multi-loop structure.

The paper is organized as follows. Section 2 introduces synchronous language SIGNAL syntax and semantics. Section 3 presents a new clock calculus method to explore the clock relations among signals. Section 4 proposes a code generation method for parallel code based on Equation-dependency Graph (Equation-dependency Graph, EDG). Section 5 illustrates the proposed method by an example. Section 6 provides related work on SIGNAL code generation. Section 7 concludes this paper.

## 2. SYNCHRONOUS THEORY AND SIGNAL

### *2.1. Synchronous Theory*

Real-time system programming models can be divided into three categories: asynchronous, synchronous, and quasi-synchronous (based on jitter/drift estimates) [12] [13]. The differences between these three models lies in the way the execution time is abstracted as a discrete sequence. The time of a reaction (reaction, i.e. input-calculation-output) is assumed to be completed within a logical moment, and the physical execution time is ignored. Thus, the synchronization mode is a platform-independent model. This section describes the related knowledge and terms of the synchronous theory briefly.

#### *2.1.1. Synchronous Hypothesis*

The synchronous hypothesis [14] is an assumption of synchronous languages: for each input, the system will complete the calculation within a bounded amount of time and then output the result before the next input. This assumption ensures that there is no overlap between reactions. Thus, the synchronous model does not consider time, but pays attention to event sequences and synchronization among events. This feature allows the user to process the system's timing problems without caring about time. The synchronization model is described in [15].

1) Logic instant and reaction

The system time in the synchronous model is abstracted as a logic instant sequence $(t_n)_{n\in N^+}$ without an overlap. In each instant, the input signal is read, computed, and output produced. Then the system enters a new global state. Each instant's execution is called a reaction.

2) Signal

Signal is a map V→$(t_n)_{n\in N^+}$ from the value to a discrete sequence. In addition to carrying data of a certain type, V can also be "⊥" and it means absent. In a logic instant, the signal value may be present or absent (⊥). A stream is the sequence of values carried by a signal.

3) Abstract clock

The clock of a signal represents the set of logical instants in which the signal varies a (non-⊥) value. A signal value can be read or computed iff the current logical instant belongs to its clock. Two signals are synchronous iff they have equal sets of instants. Clocks naturally have a logical algebraic structure.

#### *2.1.2. Mono-Clock and Multi-Clock Models*

Safety-critical real-time systems are often composed of components or sub-systems that may be deployed in a distributed environment. In designing of such systems, there are two views, mono-clock and multi-clock model. In the mono-clock model, all actions triggered in the system are controlled by a global clock. The advantage of this approach is that it is easier to specify the system and to generate code from the specification. However, the clock of each component will have a strict relationship with the global clock, resulting tight coupling among the clocks of subsystems. If the clock frequency of a component is changed, the global clock and clocks of other components need to be adjusted. Esterel and Lustre uses the mono-clock model.

In a multi-clock model, there is no global clock. Each component works according to its own clock. Components are loosely coupled so that time synchronization needs to be carried out among components. Therefore, the multi-clock model is more suitable for modeling distributed systems or those systems with a high degree of parallelism. But the multi-clock model is more complex due to synchronizations in the model, and

clock consistency needs to be maintained. Moreover, it is a little more difficult to generate code from multi-clocked specifications.

### *2.2. Introduction to SIGNAL*

SIGNAL is a declarative data-flow synchronous language developed by CNET and INRIA in France. SIGNAL adopts Polychrony multi-clock computing model [16]. Signal is the basic operation object of SIGNAL, defined as a value sequence with unlimited length and data type $(x_t)_{t\in N}$ .Given a logical instant, the signal may be present and have a value, or absent ($\perp$) and have no value. A clock is used to indicate whether there is signal $x$ in each logical instant, denoted as $\hat{x}$. In SIGNAL, a clock is represented by a signal of type event: when it exists, the value is true, otherwise it is absent. The system is specified by dataflow equations that describe relations among signal values and clocks.

SIGNAL provides four primitive constructs to describe dataflow equations, including instantaneous relation, delay, under-sampling and deterministic merging (referred to merge). Table 1 shows the syntax and semantics of four primitive constructs. It should be noted: 1) The f in the instantaneous relational operation can be algebra or Boolean operation; 2) As the special meaning of delay operations, it is also called memory equation while the left value signal of the equation is called memory signal or status signal.

SIGNAL shows data dependencies among signals and clock relations between signals, as shown in Table 2. For instantaneous relationship, the equation implicates that the left-hand signal and right-hand signals have the same clock, and this means signals are synchronous. Delay operation has the same clock relationship. These two operations are also known as synchronous operations. For the under-sampling operation, the clock of y is true if and only if the clock of x is true, the clock of b is true, and the value of b is true (denoted as [b]). For the deterministic merging, the clock of y is the union of x and z. As the clock of the signal in these two operations can be different, they are called multi-clocked operations.

Table 1 Syntax and Semantics of Primitive Constructs on Dataflow Equations

| name | syntax | semantics |
|---|---|---|
| Instantaneous relations | *y:=f($x_1$, $x_2$,..., $x_n$)* | When $x_1$, …, $x_n$ exist, y has a value f ($x_1$, $x_2$, …, $x_n$); otherwise y is absence. |
| Delay | *y:=x $init c* | The value of y is the value of x in the previous logic instant, the initial value is c, while x is absent, y is absence |
| Under-sampling | *y:=x when z* | When z has the value of true and the x exists, the y's value is the x's value; otherwise y is absence |
| Deterministic merging | *y:=x default z* | When x exists, y has the value of x; when x does not exist and z exists, y has the value of z; otherwise y is absence |

The data-flow equations in SIGNAL represents the relationship between signals and clocks, and the basic unit of the SIGNAL program is called process. It is composed by a set of data-flow equations. The SIGNAL also defines two basic constructions in the process: composition and local declaration. Table 3 shows the basic syntax and semantic of composition and local declaration, which are two other basic operations.

Table 2 Primitive constructs and corresponding clock relations

| syntax | Clock relation | explanation |
|---|---|---|
| *y:=f($x_1$, $x_2$,..., $x_n$)* | $\hat{y}=\hat{x}_1=\cdots=\hat{x}_n$ | All signal clocks in the equation are synchronous |
| *y:=x $ init c* | $\hat{y}=\hat{x}$ | The right value signal x and the left value signal sync |
| *y:=x when b* | $\hat{y}=\hat{x}\wedge$[b] | When b is present and the value is true and x exists, y exists |

| | | |
|---|---|---|
| *y:=x default z* | $\hat{y}=\hat{x} \vee \hat{z}$ | The clock of signal y is the union of x and z. |

Table 3 Syntax and semantics of primitive constructs in processes

| name | syntax | semantics |
|---|---|---|
| Synchronous composition | P\|Q | P and Q are processes; the behavior of P\|Q is the combination of behavior P and Q |
| Local declaration | P where t_1 s1; t_2 s2;...t_n sn;end; | P is a process, *s1* to *sn* are the signals defined in P invisible outside the process P. |

Here we can get the BNF expression of SIGNAL basic syntax, in which the data-flow equation *x:=yfz* means that the value of x is determined by the signal y, z and the operation f on them; P | Q is a combination of inter-process; P / x is the local declarations in the process:

$$P,Q ::= x := yfz \mid P \mid Q \mid P / x$$

SIGNAL also provides the clock construct "^" and memories construct "cell" and other extended constructs to simplify the expression of SIGNAL program, but all extensions constructs can be represented by the basic constructs, so the SIGNAL programs in this paper are written by the basic syntax. More information about SIGNAL can be found in [17], and its formal semantics can be found in [18].

We take a watchdog component [40] as an example to introduce the design and implementation of the Signal program. The functional block diagram is shown in Fig. 1. The main function of the watchdog detects whether the operation of the system is overtime. *Delay* is a timeout parameter. Whenever the *req* signal arrives, the watchdog starts to count down, and the clock *tick* will decrease by one for each tick. When the input signal *finish* comes, the watchdog will be reset. If the watchdog counts down to 0, the *alarm* will send a signal.

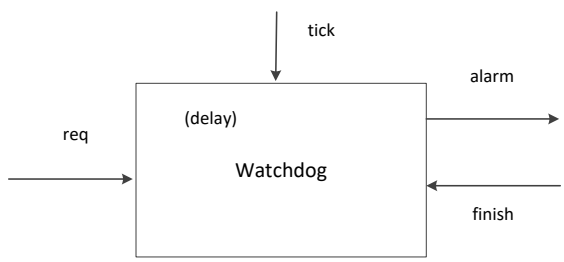



Fig. 1 Watchdog block diagram

Table 4 implements the Signal code. Line 2 defines the parameter of this process as an integer value delay. Line 3 defines the input and output signals of the program: the *req*, finish, and tick after "?" are input signals; the alarm after "!" is output signals. Lines 4 to 13 define the main body of the program, that is, the combination of data flow equations. Each equation is used to express the data and clock relationship between signals. Line 4 explicitly specifies the clock synchronization relationship between the signal hour and tick, that is, the two signals must appear or be absent synchronously at any logical moment. Line 5 defines the calculation of the hour: if *hour* exists at a certain logical instant, its value at this logical instant equals the value of its last occurrence plus 1. Line 8 defines an assignment for the signal *cnt*, using a merge operation. It means when the signal *req* exists and is true, the value of *cnt* equals the parameter delay; otherwise, if the signal finish exists

and is true, cnt is reset to -1; otherwise, when *prev_cnt* exists and is not less than 0, *cnt* equals the value of *prev_cnt* minus 1;

Table 4 Signal code of the watchdog component

```
1:process Watchdog=
2:{ integer delay;}
3:(?boolean req, finish, tick;! integer alarm;)
4:(|hour^=tick
5: |hour:=(hour$ init 0)+1
6: |cnt^=tick^=req^=finish
7: |prev_cnt := cnt $ init (-1)
8: |cnt := (delay when req)
9: default (-1 when finish)
10: default (prev_cnt -1) when ( prev_cnt >= 0)
11: default -1
12| alarm := hour when (cnt=0)
13|)
14where integer hour, cnt, prev_cnt;
15 end;
```

## 3. CLOCK CALCULUS

Clock analysis is needed to transform a multi-clocked SIGNAL program. The first problem is to extract the control structure from synchronous equations from the clock model. Clock calculus is a process to resolve the clocks and construct the control structure of the executive code, and [19] [20] proposed clock calculus for Polychrony. The idea is to extract the system of Boolean equations from the program and develop a hierarchical relation among clocks. If a single-rooted clock tree can be constructed and the sequential code can be generated based on the clock tree, the program is endochronous [16], meaning that it is deterministic w.r.t. its input-output streams. Later works, such as [21], [22] and [23] have focused on the resolution of numerical expressions to improve the precision of this determination.

Additionally, data dependencies need to be analyzed, to determine how to schedule the execution of data-flow equations. Current methods often separate clock calculus from the data-dependency analysis. In this case, two different structures are needed: one for clock hierarchy and another to represent data dependencies. To avoid the situation that hierarchical clock relations may not be consistent with data dependencies (e.g., lock-to-data cycle), the two structures need to be combined [24]. Besides, when being executed, code with hierarchical and nested control structure will be more efficient than the corresponding flat code, because, in the nested structure, guard conditions enclosed will not be checked unless the outer ones evaluate to true. Our previous work has completed part of the clock calculus research, where we designed a clock calculus that generates efficient control structures, published in [25]. Please see that paper for details, here we highlight some important definitions and algorithms of the clock calculus for a better understanding of the following content.

### *3.1. Generating Clock Equations*

The first step is to translate the data-flow equations into a system of Boolean equations called clock equations. The set of clock equations is denoted as SCE (Set of Clock Equations). The corresponding BNF is given in Table 5:

Table 5 BNF of SCE

| ClockEquation::= cl=e |
| --- |
| e::= $\hat{x}_1 \mid \diamond x_1 \mid Cond \mid \neg e \mid e \vee e \mid e \wedge e \mid e \setminus e$ |
| cl::= $\hat{x} \mid \diamond x$ |

In the definition, the left value (*cl*) of the clock equation is a signal variable, which is defined by the clock expression on the right side of the equation *(cl=e)*. Signal variables are divided into two categories. One is the clock variable $\hat{x}$, which represents the clock of signal x; the other is the value variables ($\diamond x$), which represents the value of Boolean or event typed signal x. Note that in this paper, x, the expression returning Boolean value is treated as the black box, denoted as Cond. The operators on clocks are the same as Boolean variables, including negative, union, intersection, and difference. These operators are used to represent the relations among clocks. The mappings from primitive constructs to clock equations are shown in Table 6. Note that for the convenience of the resolution, there are at most two operands in the clock equations so that two auxiliary clock variables, $\hat{z}t$ and $\widehat{xdefault}$, are introduced.

Table 6 Primitive Constructs and Corresponding Clock Equations

| Primitive | Clock relation |
| --- | --- |
| *y:=f($x_1$, $x_2$,..., $x_n$)* | $\hat{y} = \widehat{x_n}, \widehat{x_1} = \widehat{x_n}, \ldots, \widehat{x_{n-1}} = \widehat{x_n}$ (if f returns a Boolean value, add the equation $\diamond x := f(\diamond x_1, \ldots, \diamond x_n)$) |
| *y:=x $init c* | $\hat{y} = \hat{x}$ |
| *y:=x when z* | $\hat{z}t = \hat{z} \wedge \lozenge z, \hat{y} = \hat{x} \wedge \hat{z}t$ |
| *y:=x default z* | $\hat{y} = \hat{x} \vee \hat{z}$, $\widehat{xdefault} = \hat{z} \setminus \hat{x}$ |

### *3.2. Resolution of Clock Equations*

The set of clock equations generated needs to be resolved to a) find the implicit synchronization among clocks; b) get the definition for each clock; c) detect the inconsistency among clocks. This paper translates *SCE* into *SNF* (Set of Normal Form) by the resolution process. Fig. 2 illustrates the resolution process. In each iteration, every clock in *eq* will first be replaced with its definition in *SNF* (line 5, denoted as *eq* $\leftarrow$ *[SNF] eq*). If the replaced equation cannot be resolved for one of the following reasons:

1) Both sides of *eq* are complex expression (line 6);

2) LHS of *eq* exists at the *RHS* of *eq* (line 11, denoted as *eq.LHS* $\in$ *Vars (eq.RHS)*);

3) LHS of *eq* has been defined in SNF (line 11),

It will be put into *USNF* (a set storing the equations that cannot be solved temporarily). Otherwise, *LHS* of *eq* in each equation of *SNF* (denoted as *eq2*) will be replaced with its definition (*eq. RHS*) (line 16).

If a recursive definition exists after the substitution, *eq2* will be put into *USNF* (line 17-19). After that, each clock in *USNF* will be replaced with its definition in *SNF*. If both sides of the equation are equivalent, the equation will be removed from *USNF* (line 22-27). At the end of the iteration, if *eq* has not been put into *USNF*, it will be put into *SNF* (line 28).

We use OBDDs (Ordered Binary Decision Diagrams) [26] to verify the equivalent relation among expressions (as they are Boolean expressions). The replacement of variables in the algorithm can be implemented as the substitution of their OBDD values.

**Algorithm 1** clockToNF

```
1: Inputs: SCE
2: Outputs: SNF
3: SNF ← ∅, SUNF ← ∅
4: for all eq ∈ SCE do                                   ▷ e1 is the clock expression
5:     teq ← eq, eq ← [SNF]eq      ▷ replace signals in eq with its definition in SNF
6:     if both sides of eq are complex expressions then
7:         USNF ← USNF ∪ {eq}                ▷ put the translated equation into USNF
8:     else if eq.lhs is a complex expression ∧ eq.rhs is a variable then
9:         reverse(eq)                                          ▷ eq defines eq.rhs
10:    end if
11:    if eq.lhs ∈ Vars(eq.rhs) ∨ eq.lhs has been defined in SNF then          ▷
   Vars(eq.rhs) returns the set of variables in eq.rhs
12:        USNF ← USNF ∪ {eq}
13:    end if
14:    if eq ∉ USNF then
15:        for all eq2 is z = e ∈ SNF do ▷ replace eq.lhs with its definition(eq.rhs)
   in every equation of SNF
16:            teq ← eq2, eq2 ← z = [{eq}]e
17:            if z ∈ Vars(eq2.rhs) then
18:                SNF ← SNF \ {teq}, USNF ← USNF ∪ {eq2}
19:            end if
20:        end for
21:    end if
22:    TUSNF ← ∅
23:    for all eqd is e3 = e4 ∈ USNF do
24:        eqd ← [SNF]e3 = [SNF]e4
25:        if eqd.lhs and eqd.rhs are not equivalent then
26:            TUSNF ← TUSNF ∪ {eqd}
27:        end if
28:    end for
29:    USNF ← TUSNF, SNF ← SNF ∪ {eq}
30: end for
31: if NonEmpty(USNF) then
32:    error and exit the compilation
33: end if
34: return SNF
```

Fig. 2 Algorithm ClockToNF

All equations in *SCE* need to be represented by OBDDs and the time complexity of the construction will be $O(2^n)$ where n is the number of variables in the Boolean expression. Furthermore, the number of executions of the loops in line 15 to line 20 or line 22 to line 27 have the same magnitude as the size of *SCE*. As a result, the magnitude of the worst execution time will be $O(m^2 * w)$ where m is the size of *SCE* and w is the maximum execution time of the OBDD substitution. We use JDD (a Java implementation of BDD)[27] as the implementation method of OBDD. The website site shows that the performance is relatively good compared with other implementations.

After the resolution, if the *USNF* is empty, it can be deduced that there do not exist any inconsistencies or cycle definitions in the program and all equations in *SCE* have been normalized.

### *3.3. Generating Clock Equivalence Classes and SRNF*

After the resolution, a unique definition for each clock is included in *SNF* (except for clocks on the *RHS* that are undefined, usually clocks for input signals). Some of them may have identical definitions and this means they are synchronous.

The synchronous clock relation is reflexive, symmetric, and transitive so that it is an equivalence relation on the set of clocks. To get more efficient code, the paper introduces the concept of clock equivalence classes [26], that is, a partition on the set of clocks X = $\{X_i | i \in Z^+\}$. For each $X_i$, its elements $\hat{c}_1 ... \hat{c}_k$ are clock variables that are synchronous with each other. The definition of a clock equivalence class is given below.

**Definition 1** ***Clock equivalence*** *class (CEC) is a triple <ClassID, Sc, Eq>, where,*

- *ClassId: identification of the class;*
- *Sc: set of synchronous clocks belonging to this class; and*
- *Eq: actions to be executed/initiated by the clock of this class, that will be used in the construction of the clock tree*

By traversing all equations in *SNF*, clocks can be divided into these equivalent classes. For the undefined clocks, corresponding classes will be also generated. The set of clock equivalence class is denoted as *SCEC*. Note that as endochrony is the necessary and sufficient condition to generate executable code, there will be only one class for all undefined clocks. *Reduced Normal Form (RNF)* is then introduced to represent the relations among clock equivalence classes. The corresponding set of these equations is denoted as *SRNF* that can be obtained by replacing clocks with their class in *SCE*. The BNF definition is shown in Table 7:

Table 7 the BNF definition

| |
|---|
| NCE::=ClassId=e |
| e::=ClassId\|$\Diamond x$ \| $Cond$\|¬e\|e ∨ e\|e ∧ e\|e \ e |

ClasseId is the *LHS* value of the equation and *RHS* is the expression specifying relations on classes. ◊x and Cond have the same meaning as in the definition of clock equations. Note that different from *SNF*, there is no additional constraint on *SRNF* so that defined classes can exist on RHS of equations. In the remainder of the paper, *RNF* is also called as clock definition equation. *SRNF* will be used in the construction of clock trees. In the remaining sections, the term clock does not only represent the clock of signals, but also represents a clock-equivalence class: synchronous signals have the same clock represented by their equivalence classes.

### *3.4. Clock Tree Construction*

Clock equivalence classes and relations can be obtained in SRNF, from which hierarchical relations among clocks can be extracted. The definition of clock hierarchy relation " ≤" is given as follows:

- for all Boolean signals x, there are relations $\hat{x} \leq$ [x] and $\hat{x} \leq [\neg x]$;
- for variables b and c, if there are relations b≤c and c≤b, then b and c are synchronous; and
- for clock equation $b_1$=$c_1$ op c2, op∈ {∧,∨,\}, if there are relations $b_2 \leq c_1$, $b_2 \leq c_2$, then there exists relation $b_2 \leq b_1$.

This paper proposes a process to generate clock trees and a new methodology with the following features:

- Implication checking is used to insert the clock node into a deeper position
- Information of data dependency is added in the clock tree and an insertion algorithm to preserve data dependencies;

To construct a clock tree preserving the necessary properties mentioned in [25], we divide the process into three steps.

- Divide all signal equations into corresponding clock equivalence classes;
- Sort the signal equations and clock equations altogether according to the data and clock dependencies, obtaining the ordered list called *Elist*;
- Traverse *Elist* to construct the clock tree.

In the first step, data flow equations are translated into signal equations and attached to corresponding clock equivalence class. The mapping relation is shown in Table 8.

Table 8 signal definition equations for primitive constructs and input signals

| Data flow equations | Signal definition equations | Clock equivalence class |
|---|---|---|
| $y:=f(x_1, x_2,..., x_n)$ | $y:=f(x_1, x_2,..., x_n)$ | CECS[OBDD(y)] |
| y:=x when z | y=x | CECS[OBDD(y)] |
| y:=x default z | y=x | CECS[OBDD(x)] |
| y:=x default z | y=z | CECS[OBDD(z\x)] |
| y:=x $ init c | NULL | NULL |
| ? type_x x | read(x) | CECS[OBDD(x)] |

After this process, each class C in SCEC has a set of assignments (C. Eq). Furthermore, these assignments are also put into a set called *SDCE*. Along with clock equations (*RNF* in *SRNF*), these equations will be sorted to obtain the ordered list *Elist*. After the sort, *Elist* will meet the following properties:

- For signal definition equations *eq1* and *eq2*, if the *RHS* of *eq2* depends on *LHS* of *eq1*, then *eq1* precedes *eq2* in *Elist*;
- For clock definition equation Ce, C=e, and signal definition equation *eq*, if eq∈C.Eq,then Ce precedes *eq* in *Elist*;
- For clock definition equation C=C1 op C2, C1 and C2 precede C in *Elist*;
- For clock definition equation Ce, C= C1 op $\diamond x$ and signal definition *eq* x=e, then *eq* precedes Ce in *Elist.*

The construction of the clock tree is translated by the traversal of the *Elist* and equations in the *EList* will be attached to the clock node of the tree. Assume that for equation *x=e*, the node it is attached to is denoted as N, then for any node which contains operand in expression e, denoted as M, one of the relations listed below exists:

- $M=f^n(N)$, n is the positive integer;
- *f(N)=f(M)*, and *M∈LB(N)*; and
- There exists a positive integer m, $P=f^m(N)$, and *M∈LB (P).*

Fig. 3 illustrates the process of the traversal. The insertion is along with the traversal of *Elist*. If the current equation (denoted as *eq*) is a clock definition equation (**line 6**), since no node in the tree represents the class it defines, a new node is created and to be inserted into the tree (**line 7-11**). If the current equation is a signal definition equation (**line 13**), the first thing to do is to find a node to attach (**line 16**). If no proper node is found, a so-called copy-node is created. As its name indicates, there has (have) been a node (or nodes) representing the same clock but the equation is not allowed to insert into it (one of them) due to the violation of properties mentioned above. As a result, the copy-node will be first inserted into the tree (**line 18-22**), then eq will be attached to it (**line 23**).

After getting the path, the clock node will be inserted into the position at the rightmost position of the path. To insert as deep as possible, this paper uses the *clock-implication checking* based on the Breath-First Search algorithm, as shown in Fig. 4.

First, the root is put into *Queue*. Then, the iteration begins: get the head of that sub-tree; for each direct children of the head, if it is in the right side of the path (*node* $\in$ *RST*) and the clock of *CN* implies the clock of node (*CN.Clock* → *node.Clock*), put it into the tail of *Queue*. The iteration will not stop until *Queue* is empty and *CN* will be inserted at the last node of ordered list of direct children of the element got from *Queue*. Clocks relationship and data dependences can be obtained using the algorithm above. It is the basis for generating concurrent code.

**Algorithm 2** treeConstruction

**Inputs:** $CN$ ▷ denote $CN$ as the node to be inserted
**Outputs:** $head$ ▷ return node to insert
$Queue \leftarrow addNode(Root, Queue)$ ▷ put the root node into the Queue
$Vr \leftarrow VS \cup Vr$ ▷ put $Vr$ into $VS$
**for all** $eq \in Elist$ **do**
  **if** *eq is the clock definition equation* **then**
    $RST \leftarrow genPath(eq.rhs)$ ▷ obtain the sub tree on the right side of the dependency path
    $Vf \leftarrow findInsertClock(RST, eq.lhs)$ ▷ find the proper node to insert
    $Vc \leftarrow createNode(eq.lhs)$ ▷ create a new clock node for the clock
    $buildRelation(Vf, Vc)$ ▷ insert $Vc$ at the rightmost position of the list of $Vf$' s children
    $VS \leftarrow VS \cup Vc$
    $Vc.rnf \leftarrow eq$ ▷ eq is designated as the rnf of $Vc$
  **else** ▷ eq is the signal definition equation
    *find corresponding clock equivalence class of eq, denoted as C*
    $RST \leftarrow genPath(eq.rhs \cup C)$
    $Vc \leftarrow findInsertData(RST, C)$ ▷ find the proper node to insert $eq$
    **if** $Vc = NULL$ **then** ▷ if no node meeting the data and clock dependency relation is found
      $Vf \leftarrow findInsertClock(path, C)$ ▷ find a proper node to insert the clock node
      $Vc \leftarrow createNode(C)$ ▷ create the copy-node $Vc$
      $buildRelation(Vf, Vc)$
      $Vs \leftarrow VS \cup Vc$
    **end if**
    $Vc.As \leq Vc.As \cup eq$ ▷ attach $eq$ to $Vc$
  **end if**
**end for**
**return** $VS$

Fig. 3 Algorithm treeConstruction

**Algorithm 3** findInsertClock

**Inputs:** $Elist, SCEC$
**Outputs:** $VS$ ▷ clock tree
$Vr \leftarrow createNode(R)$ ▷ create clock node for the root clock
**while** $nonEmpty(Queue)$ **do** ▷ Queue is not empty
  $head \leftarrow deQueue(Queue)$ ▷ get the head from $Queue$
  **for all** $node \in direct\ children\ of\ head$ **do**
    **if** $node \in RST \wedge CN.Class \rightarrow node.Class$ **then** ▷ node is in $RST$ and the implication relation holds
      $Queue \leftarrow addNode(node, Queue)$ ▷ put node into $Queue$
    **end if**
  **end for**
**end while**
**return** $head$

Fig. 4 Algorithm "findInsertClock"

## 4. Generation of Concurrent Code based on EDG

SIGNAL describes the functional behavior of the system through data-flow equations. If equations have no data and clock dependencies, they can be executed in parallel. This paper proposes a method based on EDG that can discover the implicit parallelism and generate OpenMP parallel simulation code based on the EDG analysis. Compared with the method proposed in [28], this paper takes clock information into account. The overall process flows shown in Fig. 5. SIGNAL specifications include signal data dependencies and implicit relationships among the signal clocks. An EDG can be obtained through data dependence analysis. Clock status

equation can be obtained by analyzing clock relations. Then one can get the task sequences through the EDG division of parallel tasks and combination of clock relations information. At last, the task sequences will be mapped the grammatical structure of OpenMP.

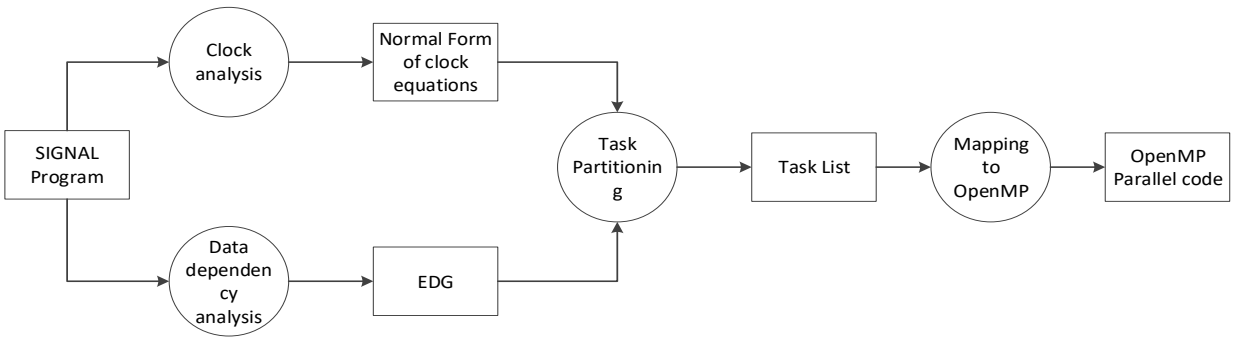



Fig. 5 Process of Parallel-Code Generation from SIGNAL Specification to OpenMP

### *4.1. Definitions and Construction of EDG*

To support SIGNAL code generation, [29] first proposed the concept of synchronous flow dependency graph (SFDG). The SDFG comprises a directed-acyclic graph (DAG), while the nodes in DAG represent signals, and the edges in DAG represent the dependencies between the signals. The appearance of the nodes and edges are determined by activation clocks. However, *SFDG* does not contain the information about how the signal is calculated based on the data dependencies. EDGs describe the dependencies among equations better to facility ate that final mapping. The data-flow equations are nodes in the EDG graph and the edges represent the data-dependency relation. A node in the EDG, denoted as NDG, can be defined as follows.

**Definitions 4**: ***A node NDG*** *of the EDG is a triple <B, L, R>, where*

- *B is a tuple <G, A>, where G is the guard condition and A is the action, meaning that the action A will be executed under the trigger of guard condition of G (A is also referred as the signal defined equation).*
- *L is the left-hand signal of action A.*
- *R is the set of right-hand signals of action A.*

Based on the above definitions, Table 9 gives the mapping rules from four primitive operations and the input signals to the corresponding NDG.

For an instantaneous relationship, y corresponds to the node L; the right value set $x_1$, …, $x_n$ corresponds to the R; the G in B is the clock C_y of signal y, action A is a relational operator $y:=f(x_1, x_2,..., x_n)$. As y and signal $x_1$ to $x_n$ are synchronized, while the clock of y is true, $x_1$ to $x_n$ must exist.

Table 9 Mapping from primitive constructs to NDG

| Syntax | L | R | G |
|---|---|---|---|
| $y:=f(x_1, x_2,..., x_n)$ | y | x1,….,xn | if C_y then y:=f(x1,…,xn) |
| y:=x $ init c | NULL | NULL | NULL |
| y:=x when b | y | x,z | if C_y then y:=x |
| y:=x default z | y | x | if C_x then y:=x |
| y:=x default z | y | z | if C_z\C_x then y:=z |
| Input signal s | S | NULL | if C_s then read(s) |

For an under-sampling operation, y corresponds to the node L; the right value x and z corresponding to R, the G in B is the clock *C_y* of signal y, the action is an assignment operation y:=x. If the action y:=x is to be executed, the value of x and z have to been obtained and the value of *C_y* is true first.

For the deterministic merging operation, we generate two nodes, corresponding to the choice of x or z. When the clock *C_x* of x is true, x is assigned to y. When the clock of x is false and the clock of z is true, z is assigned to y. Since the guard conditions of two nodes action are exclusive, it is not necessary to consider the issue of the shared variable update.

For the input signal NDG, since the input signal will get value from the environment, R is null. It will be a read operation when the clock of the L signal is true.

The EDG is a directed acyclic graph that contains the global data dependency of the SIGNAL program:

**Definition 5:** ***EDG is*** *a tuple <SNDG, →>, where*

- *SNDG is a set of nodes NDG.*
- *→ is the dependency between the nodes: if the node v1, v2 satisfy the relation v1→v2 if and only if v1. L ∈ v2. R.*

The *SNDG* can be constructed by analyzing the collected data-flow equations, where the guard condition clock of each node denoted by the corresponding clock equivalence class. The EDG can be obtained by analyzing the data relation between the *SNDG's* right value and left value. It should be noted that, for the case, that the memory signal appears in the equation's *RHS*, because of the peculiar semantic of the delay operation, we can ensure that the data dependency and the guard condition G in the node are satisfied so that the value obtained from the memory signal is correct.

### *4.2. Parallel Task Partitioning based on EDG*

Through analyzing the EDG to collect data dependencies among equations, one may classify it into parallel-execution and serial-execution parts. First, one can divide EDG into parallel tasks by a topological sort, then, put the clock information into those parallel tasks.

#### *4.2.1. EDG Division based on Topological Sorting*

In the EDG, the binary relation "→" defines the dependency between the nodes, that is, for two nodes v1, v2, if the left value signal of v1 corresponding to defined equation eq1 appears in the right of v2 corresponding to defined equation *eq2*, then *eq2* relies on *eq1*. As the data dependency satisfies strict partial order, we can define its transitive closure, denoted "↘". If the two data-flow equations *eq1* and *eq2* in the program do not have this relationship, then these two equations can be executed in parallel. To generate parallel code from the SIGNAL specification, we need to divide the EDG further. Here one provides the definition of parallel tasks.

**Definition 6:** ***A parallel task*** *is a tuple <T, ↗>, where*

- *T is a partition of the nodes in the EDG, that is*
- *for any t ∈T, t ∈ EDG.SNDG,*
- *For two any nodes $t_1$、$t_2$ ∈T, $t_1 \cap t_2 = \emptyset$,*
- *The union of all the elements in T is the node set of the EDG. An element t is called as task node.*
- *"↗" is the binary precedence relation on T.*

The task partition method has a great impact on the final parallel program performance on different target platforms. We consider a topological sorting method to divide the EDG. The key is to define a partition of T and a priority relation from the dependence relation:

- For any t∈T, t is defined as an anti-chain on "↘", that is, for any two nodes n1, n2∈t, n1↘n2 and n2↘n1 are always false.
- For any t1, t2∈T, t1↗t2 if and only if there is at least one node n in t2 and a node m in t1: m↘n.

Table 10 Partition of EDG based on topological sorting

```
1:Input: EDG
2:Output:TaskList % ordered table, the order shows linear relationship
3:NodeSet ← EDG.Nodes % the initial value NodeSet contains all the nodes in EDG
4:while NodeSet≠∅
5:    setNoDegree←findNodeWithNoIndegree(NodeSet) % find the node with nodegree in NodeSet
6:    tempTask←createT(setNoDegree) % Establish the elements t in the Task
7:    addTask(TaskList,tempTask) %add the new build element to the end of the queue
8:    removeRelation(NodeSet,setNoDegree) %delete the binary relations associated with the nodes in the setNoDegree set
9:    removeNode(NodeSet,setNoDegree) % delete all the nodes in setNodegree for the NodeSet
10:endwhile
11:return TaskList
```

Table 10 gives the EDG partition algorithm, the input of the algorithm is the EDG, and the output is the *TaskList* that shows an ordered list of the parallel tasks. First of all, the algorithm puts all the nodes in the EDG into the *NodeSet* (line 3), then starts the iterative process (line 4). Line 5 to line 10 describe the topological sort: first, it finds the nodes without in-degree in the *NodeSet* (line 5, the in-degree refers to a data dependency "→") to form a node set named *setNoDegree* that is an element of the Task, denoted *tempTask* (line 6). Then, it adds the *tempTask* into the *TaskList* (line 7) and delete all nodes in *setNoDegree* set and relations out-degree with those nodes in *NodeSet* (line 8 and line 9). The iteration ends when the *NodeSet* is empty, that is, when all nodes in the EDG have been partitioned into the *TaskList*.

For any EDG, the sequence *TaskList* = $\{t_{n_1},..., t_{n_k}\}$obtained by the above topological sort satisfies the following properties:

- Any t∈T is an anti-chain of the dependency relation"↘"
- For any integer x, y∈ $\{n_1, ..., n_k\}$, $t_x$ and $t_y$ satisfy the relation $t_x↗t_y$ or $t_y↗t_x$.
- For any integers x, y ∈ {1, ..., k}, if x <y, for any node n∈ $t_{n_y}$, there is no node m∈ $t_{n_x}$, such n↘m established.

The *TaskList* is a special case in the parallel task, while the priority relationship "↗" is a total order relationship among the task nodes. We can further define the adjacent relationship between the task list: for the task sequence *TaskList*= $\{t_{n_1},..., t_{n_k}\}$ and x, y∈$\{n_1, ..., n_k\}$, if y = x + 1, then the task node $t_x$ is left adjacent of the task node $t_y$, denoted $t_x \Rightarrow t_y$. The above algorithm is applicable to the case that there is no circular dependency in the EDG. If there is a signal s meet "s↘s", the EDG cannot be partitioned. For multi-clock operations, it should be noted the data dependencies depend on a clock.

Hence, for a data dependency $n_1 \xrightarrow{c_1} n_2 \xrightarrow{c_2} ... \xrightarrow{c_{k-1}} n_k$ , if the value of $c_1 \wedge c_2 .... \wedge c_{k-1}$ is the empty clock 0, then the cyclic dependency will never be established. The algorithm assumes that there are no such cyclic dependency or pseudo-cyclic dependency in the program.

### *4.2.2. Combination of Clock Information and Parallel Tasks*

Through the EDG task partition, one can get the task sequence *TaskList* where elements can be executed according to the sequence, and the nodes in the inner of the element without data dependency can be executed in parallel. However, this structure lacks information on how to calculate the clocks, so it is necessary to put the clock equations (called *RNFS*) into the right position of the task sequence. To this end, we give the properties that need to be satisfied:

- All the signal definitions must be executed after its activation clock has been calculated.
- All the signal definitions must be executed after the equation right value or the value variable has been calculated.

To satisfy the above properties, one adds the clock equation set into its neighbor binary relation"$\Rightarrow$" by traversing the *TaskList*, denoted as $t1 \overset{ClockDef}{\Rightarrow} t2$, where the *ClockDef* is the clock defined equation set between t1 and t2. It means that equations in the *ClockDef* must be executed before the task $t_2$, and the equation in the *ClockDef* may depend on the calculation result in $t_1$.

From the root clock, we can traversal the *TaskList* from the second element. Firstly, to get the signal definition of the current task; then find out the *RNF* set *ClockDef* from RNFS that did not add into the binary relation and relies on these signals defined equation, finally the *ClockDef* will be added into the binary relation *task.pre⇒task* to form *task.pre⇒task.*

For a task list $t_1 \Rightarrow \ldots \Rightarrow t_n$, we can get a sequence $t_1 \overset{ClockDef1}{\Rightarrow} t \cdots \overset{ClockDefn-1}{\Rightarrow} t_n$, that satisfies the following conditions:

- For any binary relation $t_k \Rightarrow t_{k+1}$, the equation calculation in the *ClockDef* does not depend on nodes $t_{k+1},\ldots, t_n$.
- The values of signal defined equation clock $C_1,\ldots, C_m$ in any $t_k$ have been calculated in the clock defined equation *ClockDef1, …, ClockDefk-1*.

The above properties will ensure that the sequence maintains the data dependency and the clock relationships in the SIGNAL program.

### *4.2.3. Mapping Parallel Tasks to OpenMP Parallel Structure*

OpenMP is a multi-platform shared-memory parallel programming API, that can support programming languages such as C, C++, FORTRAN. It provides many mechanisms including compiling guidance, application-programming interface (API), environment variables. It also supports users to describe the parallel algorithm at a high level of abstraction. Now, we previously converted SIGNAL programs into a parallel task sequences and execution actions such as simple math, logic, calculation, or assignment. Therefore, we use the compiler-guided syntax "parallel sections" of OpenMP as mapping object. The syntax is shown in Table 11:

Table 11 the compile-guided syntax “parallel sections”

```
#pragma omp sections [clause [[,] clause] ...]
{
[#pragma omp section]
structured-block
[#pragma omp section ]
structured-block
...
}
```

The guidance statement “#pragma omp sections” contains a parallel domain. The code blocks between “structured-block” in the guidance statement “pragma omp section” can run in parallel, while codes interior blocks run in serial. According to the semantics of parallel task sequences, we can get mapping rules from the task sequence elements to sections syntax, as shown in Table 12.

Table 12 Mapping from TaskList to sections

| Task sequence element | Sections syntax |
|---|---|
| Task node | #pragma omp sections{<br>} |
| Signal defined equation eq | #pragma omp section{<br>if(eq.Class==true){<br>eq.A<br>}} |
| Task list t1 $\Rightarrow$ t2 $\Rightarrow$ t3… $\Rightarrow$ tk | #pragma omp sections {<br>t1 %t1sections code<br>}<br>…<br>#pragma omp sections {<br>tk %tk sections<br>} |

Every task in the sequence is mapped to a code block encircled by “#pragma omp sections” and every signal definition equation in the task will be translated to a block encircled by “#pragma omp section”. According to the semantics of sections, all signal definition equations in the same task can be executed in parallel.

The third line in the Table 12 specifies that a task’s sequential relation is mapped to a code block entitled “sections”. Next, during compilation of the OpenMP wrapper, several threads are generated to support the parallel execution.

These threads will cost more time and resources. If there is only one node in the task, OpenMP will generate actions instead of code block sections so as to improve the program efficiency.

Apart from task and task nodes, clock definition equations also need to be mapped. In order to improve the degree of parallelism, for the binary relation $t_1 \ \ t_2$, consider clock defined equation set *ClockDef1*:

- If there is only one equation in *ClockDef1*, it will generate directly the corresponding assignment actions;
- If there are several equations and there are not data-dependency relations among equations, it will generate the parallel code block using the same proposed method.
- If there are data-dependency relations, it will generate the sequence code.

Through above mapping method, we can get OpenMP+C codes that satisfies the clock and data-dependency relations. The code can be further optimized based on the specific target platform and the OpenMP compiler. For instance, the number of threads can be parameterized to be equal to the number of section code blocks. The iteration mode of the SIGNAL compiler Polychrony is adopted to simulate program behaviors, which is unlimited circle simulation and interactive process. Every circle represents an execution process in the logic instant. We can get an OpenMP parallel structure as iteration core parts and update the value of memory signals after every iteration.

## 5. Case Study

This section uses a typical SIGNAL program to illustrate the process of generating OpenMP parallel code based on the proposed method. First, we analyze the data-flow equations to obtain clock information, then generate an EDG by analyzing the data dependency, then divide the tasks into parallel parts, and finally map the parallel task sequence to the OpenMP parallel structure.

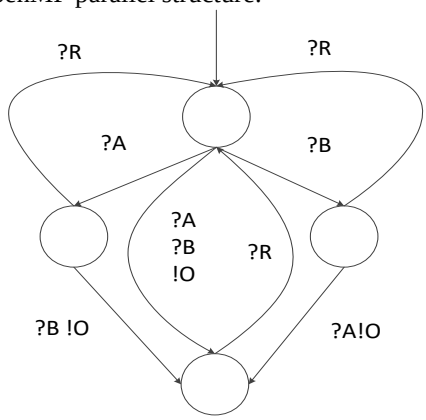



Fig. 6 State machine of ABRO

Table 13 SIGNAL Program of ABRO

```
1:process ABRO=
2:( ? boolean A, B, R;! boolean O;)
3:| A^=B^=R
4:| A^=A_received
5:|A_received^=B_received^=after_R_until_O
6:| nR := not R
7:| RT := nR when nR
8:| A_received := RT default AR
9:| AT := A when A
10:| AR := AT default Adelay
11:| Adelay := A_received $ init false
12:| BT := B when B
13:| B_received := RT default BR
14:| BR := BT default Bdelay
15:| Bdelay := B_received $ init false
16:| nO:= not O
17:| from_R_before_O := nO default RR
18:| RR := Re default after_R_until_O
19:| Re := R when R
20:| after_R_until_O := from_R_before_O $ init true
21:| O := true when ABR
22:| ABR := A_received when Arr
23:| Arr := B_received when after_R_until_O|)
24:where boolean nR, nO, A_received, B_received, from_R_before_O ,
Adelay,Bdelay,AR,BR,RR,ABR,Arr,after_R_until_O,AT,BT,RT,Re; end;
```

Table 14 Partition of clock equivalence classes for ABRO

| |
|---|
| C_2={$\hat{R}$, $\widehat{A\_received}$, $\widehat{B\_received}$, $\widehat{after\_R\_until\_O}$, $\hat{A}$, $\hat{B}$, $\hat{R}$, $\widehat{nR}$, $\widehat{RR}$, $\widehat{Adelay}$, $\widehat{Bdelay}$, $\widehat{from\_R\_before\_O}$, $\widehat{AR}$, $\widehat{BR}$} |
| C_6={$\widehat{RT}$, $\widehat{R\_false}$, $\widehat{Re}$} |
| C_75={$\widehat{A\_received\_default}$, $\widehat{B\_received\_default}$, $\widehat{RR\_default}$} |
| C_13={$\widehat{A\_true}$, $\widehat{AT}$} |
| C_26={$\widehat{B\_true}$, $\widehat{BT}$} |
| C_77={$\widehat{AR\_default}$} |
| C_79={$\widehat{BR\_default}$} |
| C_84={$\widehat{ABR}$, $\widehat{Arr\_true}$} |
| C_92={$\widehat{from\_R\_before\_O\_default}$} |
| C_82={$\widehat{after\_R\_until\_O\_true}$, $\widehat{Arr}$} |
| C_86={$\widehat{ABR\_true}$,$\widehat{nO}$, $\hat{O}$} |

### *5.1. Example SIGNAL Program*

This paper chooses the classic example ABRO [30] as the case study. ABRO automaton, as shown in Fig. 6. ABRO has three input signals, A, B, R, and an output signal O. In the initial state ABRO waiting for the input signal, when the A, B signal has read in any order, ABRO output signal O. Signal R is used for resetting ABRO state:

(1) Once O is outputted, ABRO will not receive the input from either A or B until it receives R and returns to the initial state;

(2) If R is received after one of A and B has been received, ABRO will return to the initial state, waiting for the next inputs of A and B.

According to the feature and function of ABRO, [15] gives the corresponding SIGNAL program, and a modified version is given in Table 13. The basic unit of the Signal program is a process (**line 1**). Line 2 defines the declaration of program input signals (A, B, R) and the output signal (O). In order to generate simulation code, set the input signals A, B, and R to Boolean type, and establish the synchronization relationship of the three signals (**line 3**). This synchronization condition guarantees that the three signals can always be read in at the same logical moment. Besides, when the program reads the value of the input signal as true, it means that the signal can trigger a state transition. The output signal O is also of Boolean type. If the calculated value is true, the signal can be output. Lines 3 to 5 specify the clock synchronization relationship between signals. Because the signal synchronization relationship is transitive, the signals *A, B, R, A_received, B_received,* and *after_R_until_O* are synchronized. In the code, *Adelay* and *Bdelay* represent the value of the signal which A and B received last time so that it can be guaranteed that A and B do not appear in a tick at the same time (A is before B or B is before A), and when there is no R signal, the value of O can still be calculated.

### *5.2. Exploration of Clock Relations*

During the translation, each signal x corresponds to a clock variable $\hat{x}$. For sampling operations and merging operations, temporary clock variables are generated correspondingly. For example, for the equation "*RT:=nR when nR*", the corresponding clock variable *R_false* is generated, and its value is defined as " $\hat{R}$ ⩘ *nR*", R is the value variable of the signal R, which means the value of the signal variable. *R_false* is true when the signal R exists and its value is false. For the merge operation "*AR:=AT default Adelay*", the clock variable *AR_default* and its definition equation " $\widehat{AR\_default} = \widehat{Adelay} \setminus \widehat{AT}$" are generated. After obtaining the set of clock equations, it can be parsed to get the clock equivalence classes. Table 14 shows the partition results of the clock

variable equivalence classes. Each integer x in equivalence class "C_x" is the value of BDD. The equivalence classes C_2 is the root clock of the program. Table 15 shows the corresponding *RNFS* equations. Since only the clock C_2 appears on the right equation, the program meets the endochrony property and C_2 is the root clock of the program.

Table 15 RNFS of ABRO

| |
|---|
| C_6=C_2∧R |
| C_75=C_2 \ C_6 |
| C_13=C_2∧A |
| C_77=C_2 \ C_13 |
| C_26=C_2∧B |
| C_79=C_2 \ C_26 |
| C_82=C_2∧after_R_until_O |
| C_84=C_82∧Arr |
| C_86=C_84∧ABR |
| C_92=C_2 \ C_86 |

### *5.3. Generation of EDG and Task Division*

By analyzing the ABRO data-flow equations and data dependencies, one can get its EDG as shown in Fig. 7. Although the EDG in the graph does not add the clock information, the clock relations have been taken into consideration when analyzing data dependency. For example, for the equation *ABR: =A_received* when *Arr*, the signal defined equation *ABR=A_received* depends on the value of *A_received* and Arr. In addition, for the merge operation, two mutually exclusive actions are put into a node and will not be executed simultaneously in same iteration.

From this EDG, a parallel task sequence will be generated by topological sort and by adding the clock information as shown in Fig. 8. The solid line arrow in the figure represents the linear relationship of the parallel task execution, and the equations in a task can be performed in parallel.

### *5.4. Generation of Parallel Code of OpenMP*

Table 16 shows the generated OpenMP parallel code fragments where the three input signals A, B and R can be simultaneously read. The generated code corresponds to the OpenMP parallel structure.

### *5.5. Validation of Generated Code*

A SIGNAL code generation tool was developed as a plugin tool on the Eclipse platform. The generated simulation code is available on the GitHub website [39].

The procedure obtains the input signal by reading a text file and outputs to the corresponding text file. In the ABRO program example, the input signals are A, B, R, the output signal is O. The corresponding signal values of the three input signal files are respectively “1 1 1 1 0 1 0”, “1 1 1 1 1 0 1” and “0 0 1 0 1 0 0”. We obtain the output file O.txt which is “1 0 0 1 0 1 0”, which illustrates that the generated parallel code can effectively simulate the functional behavior of the ABRO program.

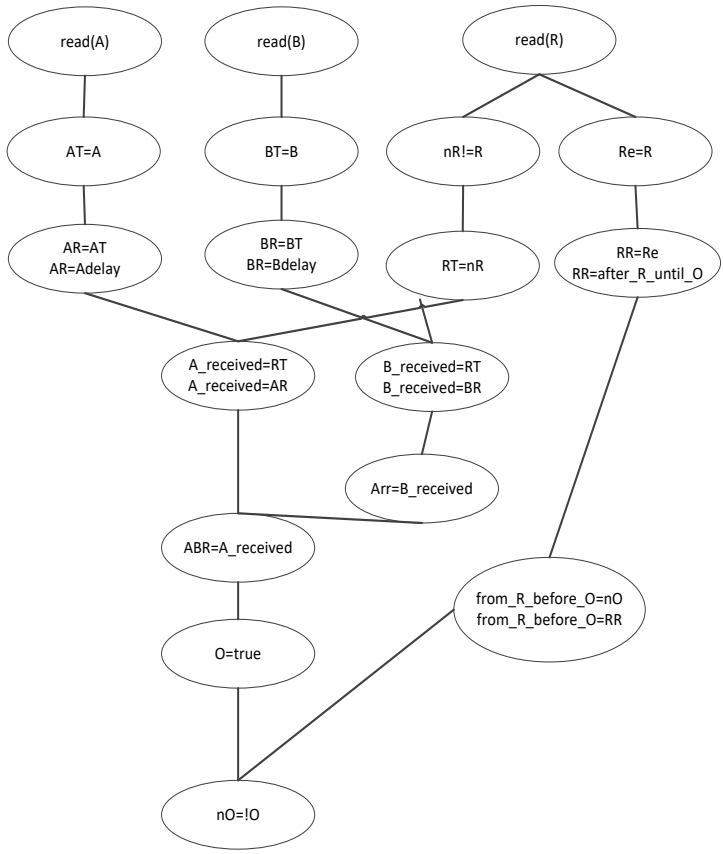



Fig. 7 EDG of ABRO

Table 17 shows a possible result of the ABRO program once executed. For example, for the logic instant t1, the signal A and B are true at the same time and the R is false, then the output signal O is true. However, although A and B are true at the logic instant t2, the state is not reset at this time, the value of O is not existing. In logic instant t3, the input value of R is true and the state has been reset (the value of A, B are not still read in, the value of O does not exist). In logic instant t4, the value of O is true after A and B have been read at the same time. At instant t5, R reset the value of O at the instant t4, and the value of B is stored in *B_received*. At instant t6, the value of A is 1 and the value of R is 0, so it outputs 1.

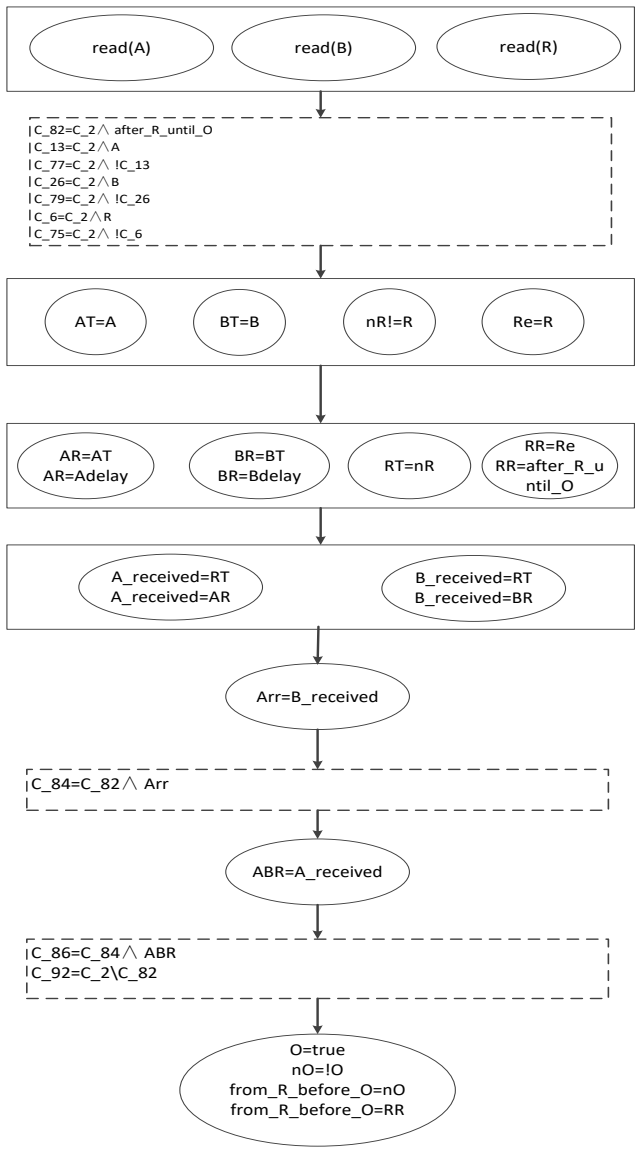



Fig. 8 TaskList of ABRO

Table 16 Fragment of OpenMP code

```
1:int core(){
2:          int mark=0;
3:          #pragma omp parallel sections
4:          {
5:                    #pragma omp section
6:                    {
7:                              if(read_A()==EOF)
8:                                        mark=1;
9:                    }
10:                   #pragma omp section
11:                   {
12:                             if(read_B()==EOF)
13:                                       mark=1;
14:                   }
15:                   #pragma omp section
16:                   {
17:                             if(read_R()==EOF)
18:                                       mark=1;
19:                   }
20:         }
…
```

Table 17 A Possible Execution Trace of ABRO

| Signal | t1 | t2 | t3 | t4 | t5 | t6 | t7 |
|---|---|---|---|---|---|---|---|
| A | 1 | 1 | 1 | 1 | 0 | 1 | 0 |
| B | 1 | 1 | 1 | 1 | 1 | 0 | 1 |
| R | 0 | 0 | 1 | 0 | 1 | 0 | 0 |
| O | 1 | 0 | 0 | 1 | 0 | 1 | 0 |

### *5.6. Performance comparison*

To compare the performance of the OpenMP parallel code generated by our method with the serial code generated by Polychrony, we test the execution times of the two separately. Firstly, we establish the running environment with CodeBlocks of version 20.03 such as configuring it to support OpenMP. Our experiment is divided into four steps, and in each step, we run 10 times and remove the strange data, and then get the average to reduce the error.

1) In the first step, we compare directly the execution time between two codes, and the result for two parallel and serial codes is 5*ms* and 1*ms* respectively. This is because the generated codes are simple and don't contain many cycles;
2) In the second step, we add a loop to simulate the situation of reducing the speed of reading the input file A.txt which is very likely to happen in the actual system like blocking time-consuming IO operations The loop structure is shown in Table 18, where we also print out the thread number to verify the parallel calculation. The result is 227*ms* and 505*ms* respectively;

Table 18 Function for reading files

```
int read_A(){
```

```
        for(int i=0;i<100;i++){
          printf("section A, thread id=%d\n", omp_get_thread_num());
        }
        return (fscanf(fp_A,"%d",&A));
}
```

3) In the third step, we also add a loop in *read_B()* function, and the result is 446*ms* and 976*ms* respectively;
4) In the last step, we continue to add a loop in *read_C()* function, and the result is 672*ms* and 1452*ms* respectively.

As shown in Fig. 9, it can be concluded that the parallel code shows superior performance under complex multi-loop structures when compared with serial code. Because in the parallel code, these three functions that read files are called in parallel while they are executed sequentially in the serial code, as shown in Table 16.

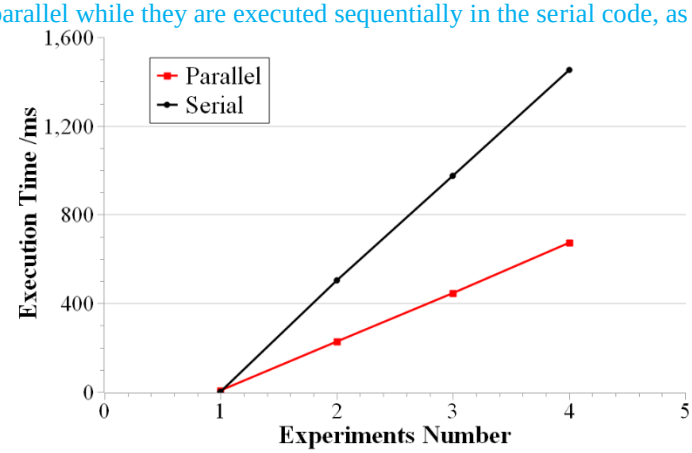



Fig. 9 Performance comparison graph

## 6. Related Work

Code generation from SIGNAL specifications, particularly generating multi-threaded and distributed code, has been an active research area, but the work on cross-platform code generation is at an early stage.

There are two main Signal compilers working in two ways. One is developed by the Geensys which is integrated RT-Builder as code generation tool. Another is developed by the ESPRESSO INRIA project team called Polychrony. Polychrony compiler provides the Signal language into C, C ++, Java and other languages serial code. The tool is integrated as an Eclipse plug-in in the TOPCASED environment within the Polarsys Industry Working Group. Users can create a Signal model then convert to needed code by using plug-ins. Polychrony proposes a directed acyclic Hierarchical Conditional Dependency Graph (HCDG) to describe clock relations of hierarchy and the respective signal calculations. However, it requires calculating clock relations and scheduling relationship respectively and then merging them into a diagram HCDG. It does not optimize clock calculation or clock hierarchy as we do but proposes several clustering policies. How to optimize the structure of the clock tree to generate optimized code is this paper's objective.

Besnard et al. [31] introduce a multi-threaded code generation for SIGNAL supporting either static or dynamic scheduling. For static scheduling, the compiler generates multiple clusters of code according to a scheduling diagram. There is a main computation cluster used for iteration. Each cluster has its own single clock tree. In the distributed code generation from the dynamic scheduling, each equation that can be executed in parallel in the SIGNAL program will generate a thread. The micro-thread waits for input signal and sends notify

signal when outputting. Such unrelated processes can be executed in parallel. However, the multi-threaded code is generated using the particular thread library of OepnMP, so it cannot be executed in a different platform.

In a distributed or multi-core system, due to the communication delay between components and some other reasons, signals between components cannot meet the synchronization relationship. These systems are called global asynchronous locally synchronous (GALS) systems and do not satisfy the endochrony property. Potop-Butucaru [32] proposed a weak endochrony theory. As the name implies, weak endochrony reduced conditions of SIGNAL programs to generate deterministic code, that the program can exist multiple root clocks. If the inter-signal meets the full-diamond conditions, it can generate deterministic multi-threaded code. However, how to detect whether the program can satisfy the property of weak endochrony is difficult. Potop-Butucaru [33] proposed a method to detect weak endochrony nature and gave the conflict-free minimum reaction collection from the SIGNAL programs. If this collection is unique, the program is weakly endochronous. But a disadvantage is that one needs to translate the SIGNAL program to a stateless abstraction, which makes some weakly endochronous programs lose information so that they cannot satisfy this property.

Talpin et al. [34] proposed a method to check weak endochrony based on bounded model checking and isochrony [35] property. However, these two methods cannot be applied to all weakly endochronous programs. Jose [36] proposed a multi-threaded code generation method based on synchronous data-flow dependency graph (SFDG), but does not give a verification method for the weak endochrony property.

Compared with the previous multi-threaded code generation method based on weak endochrony, we obtain a good degree of parallelism by using the EDG description and the partition of tasks.

In addition, Baudisch [37] proposed a method from the Synchronous Guarded Actions (SGA) to generate the OpenMP parallel code. SGA is first converted to the Dependency Graph (DG). Then the synchronized parts can be obtained from the DG from which the OpenMP structure can be finally generated. However, since SIGNAL has great differences with the synchronization guard action in syntax and semantics, especially in the semantics of the clock and response action [38], the proposed approach is very different from [37], especially in the clock analysis. SGA uses the mono-clock model, without considering the clock access to information. This paper chooses the multi-clocked SIGNAL model and proposes a method that adds clock information to the parallel task sequence. In addition, for the intermediate data structure design, the EDG can describe the intuitive global data dependencies of the program, better than the bipartite graph representation method proposed in [37].

Compared with the previous studies, the proposed method has the following characteristics:

- It adopts an EDG that can describe data dependencies between equations and the equations in the SIGNAL program. It can extract the parallel parts easily by the topological sorting. The generated code can get better parallelism.
- The paper uses a cross-platform parallel programming techniques OpenMP as the generated object. The generated parallel simulation code can be executed on a variety of platforms with great flexibility.

## 7. Conclusion and Future Work

This paper presents a parallel code generation method from SIGNAL synchronization specification to OpenMP with clock calculus that can extract the clock information from the SIGNAL program based on analysis of Boolean equations. Then, after that, EDG is used for describing the global data-dependency relationship in the program, and the topological sorting is used to allocate tasks to be executed in parallel. Finally, the parallel task sequence is mapped to the sections structure of the OpenMP.

Future work will focus on the following three aspects:

1) Optimize the parallel task partitioning, and improve the parallelism and execution efficiency of the generated code.

2) Formalize the code generation process and make the process modular. Each part will be formalized by using Coq and Caml to verify correctness and validate the whole process to ensure the correctness of the entire compilation process.

3) Study the weak endochrony theory and investigate feasible algorithm to verify whether SIGNAL programs are weakly endochronous. The goal is to generate parallel code from the weakly endochronous programs.

## ACKNOWLEDGMENTS

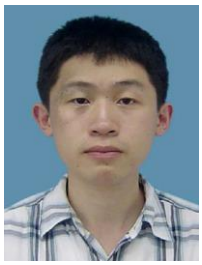
This work was partially supported by the National Natural Science Foundation of China under Grant 61672074, 61672075, Project of National Key Research and Development of China under Grant 2018YFB1402702, Funding of Ministry of Education and China Mobile MCM20180104, State Key Laboratory of Software Development Environment (No. SKLSDE-2020ZX-21).

## REFERENCES

[1] Knight JC. Safety Critical Systems: Challenges and Directions. Proc. of the 24th International Conference on Software Engineering. Orlando, 2002. 547-550.
[2] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard Version 3.0. http://www.mpi-forum.org/docs/docs.html
[3] The OpenMP API specification for parallel programming. http://openmp.org/wp/
[4] Intel Thread Building Blocks. http://www.threadingbuildingblocks.org/
[5] Benveniste A, Gérard B. The synchronous approach to reactive and real-time systems. Proceedings of the IEEE, 1991. 1270-1282.
[6] Benveniste A, Caspi P, Edwards SA, et al. The Synchronous Languages 12 Years Later. Proceedings of the IEEE,2003, 91(1): 64-83.
[7] Berry G, Gonthier G. The ESTEREL synchronous programming language: design, semantics, implementation 1992, Sci. Comput. Program, 19(2):87–152.
[8] Halbwachs N, Caspi P., Raymond P, et al. The synchronous data flow programming language LUSTRE. 1991, Proceedings of the IEEE, 79(9):1305-1320.
[9] Le Guernic P, Gautier T, Le Borgne M, et al. Programming real-time applications with SIGNAL. 1991, Proceedings of the IEEE, 79(9):1321-1336.
[10] Schneider K, The synchronous programming language QUARTZ, Tech. rep., Internal Report 375, Department of Computer Science, University of Kaiserslautern, Kaiserslautern, Germany (2009).
[11] Polychrony, http://www.irisa.fr/espresso/Polychrony/
[12] Kirsch CM. Principles of real-time programming. 2002 Conference on Embedded Software. Springer Berlin Heidelberg, 2002. 61-75.
[13] Jantsch A, Sander I. Models of computation and languages for embedded system design. Computers and Digital Techniques, IEE Proceedings-. IET. 2005. 114-129.
[14] Potop-Butucaru D, de Simone R, Talpin JP. The synchronous hypothesis and synchronous languages, http://pdf.aminer.org/000/213/379/modeling_distributed_embedded_systems_in_multi-clock_esterel.pdf, 2005
[15] Gamati A. Designing embedded systems with the Signal programming language: synchronous, reactive specification. Springer Publishing Company, Incorporated, 2009.
[16] Le Guernic P, Talpin JP, Le Lann JC. Polychrony for System Design. Journal for Circuits, Systems and Computers, 2003,12(3):261-303.
[17] Besnard L, Gautier T, Le Guernic P. Signal v4-Inria version: Reference manual, http://www.irisa.fr/espresso/Polychrony/document/V4_def.pdf
[18] Yang Z, Bodeveix JP, Filali. M. A comparative study of two formal semantics of the SIGNAL language. Frontiers of Computer Science, 2013, 7(5): 673-693.
[19] T.P. Amagbegnon, L. Besnard, P. Le Guernic, et al., Arborescent canonical form of boolean expressions, Tech. Rep. 2290, INRIA, 1994.
[20] P. Amagbégnon, L. Besnard, P. Le Guernic, Implementation of the data-flow synchronous language signal, ACM SIGPLAN Notices 30 (6) (1995) 163 – 173.

[21] A. Gamatié, T. Gautier, P. Le Guernic, Toward static analysis of SIGNAL programs using interval techniques, in: Synchronous Languages, Applications, and Programming, Vienna, Autriche, 2006 (SLAP 2006). <http://hal.archives-ouvertes.fr/hal-00544123>.
[22] P. Feautrier, A. Gamatié, L. Gonnord, Enhancing the Compilation of Synchronous Dataflow Programs with a Combined Numerical-Boolean Abstraction,Tech. Rep. 2nd Version, July 2013. <http://hal.archives-ouvertes.fr/hal-00780521>.
[23] M. Nebut, Specification and analysis of synchronous reactions, Formal Aspects Comput. 16 (3) (2004) 263 – 291.
[24] L. Besnard, T. Gautier, J.-P. Talpin, Code generation strategies in the Polychrony environment, Tech. Rep. RR-6894, INRIA, 2009. <http://hal.inria.fr/inria-00372412>.
[25] Kai Hu, Teng Zhang, Zhibin Yang, Wei-Tek Tsai. Simulation of real-time systems with clock calculus[J], Elsevier： Simulation Modelling Practice and Theory, Vol 51: pp. 69-86, 2015,
[26] J.-P. Talpin, J. Ouy, T. Gautier, L. Besnard, P. Le Guernic, Compositional design of isochronous systems, Sci. Comput. Programm. 77 (2) (2012) 113 – 128.
[27] https://bitbucket.org/vahidi/jdd/src/master/.[OL]
[28] Hu K, Zhang T, Yang Z. Multi-threaded Code generation from Signal to OpenMP. Frontiers of Computer Science, 2013, 7(5):617-626.
[29] Maffeïs O, Le Guernic P. Combining Dependability with Architectural Adaptability by means of the SIGNAL Language. Static Analysis. Springer Berlin Heidelberg, 1993. 99-110.
[30] Berry G. The foundations of esterel. in: Proof, Language, and Interaction, 2000. 425-454.
[31] Besnard L, Gautier T, Talpin JP. Code generation strategies in the Polychrony environmen. http://hal.inria.fr/docs/00/37/24/12/PDF/RR-6894.pdf
[32] Potop-Butucaru D, Caillaud B, Benveniste A. Concurrency in synchronous systems. Formal Methods in System Design, 2006, 28(2): 111-130.
[33] Potop-Butucaru D, et al. From concurrent multi-clock programs to deterministic asynchronous implementations. Fundamenta Informaticae,2011,108(1): 91-118.
[34] Talpin JP, Ouy J, Gautier T, et al. Compositional design of isochronous systems. Science of Computer Programming, 2012, 77(2): 113-128.
[35] Benveniste A, Caillaud B, Le Guernic P. Compositionality in dataflow synchronous languages: Specification and distributed code generation. Information and Computation, 2000, 163(1): 125-171.
[36] Jose BA, Shukla SK, Patel HD, et al. On the deterministic multi-threaded software synthesis from polychronous specifications. 6th ACM & IEEE International Conference on Formal Methods and Models for Co-Design (MEMOCODE 2008), 2008. 129-138.
[37] Baudisch D, Brandt J, Schneider K. Multithreaded code from synchronous programs: Extracting independent threads for OpenMP. Conference on Design, Automation and Test in Europe. European Design and Automation Association, 2010. 949-952.
[38] Yongwang Zhao, David Sann, Fuyuan Zhang, Yang Liu, "Formal Specification and Analysis of Partitioning Operating Systems by Integrating Ontology and Refinement", IEEE Transactions on Industrial Informatics, Volume 12, No. 4, August, 2016, pp. 1321 – 1331
[39] https://github.com/FISHzj/Signal-Code.[OL]
[40] Gamati A. Designing embedded systems with the Signal programming language: synchronous, reactive specification [M]. Springer Publishing Company, Incorporated, 2009.

**Kai HU** is a professor at Beihang University, China. He received his Ph.D. degree from Beihang University in 2001. From 2001 to 2004, he did the 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), Beihang University. His research interests concern embedded real time systems and high-performance computing. He has good cooperation with IRIT and INRIA Institute of France on study of AADL and synchronous languages.
Email: hukai@buaa.edu.cn



**Teng Zhang** received his B.E. in computer science and engineering from Beihang University in 2011. He is now the master's degree student at the same university. His research interests include synchronous languages, modeling of embedded system and formal methods.
Now he is a PH.D student, University of Pennsylvania, USA
Email: tengz@seas.upenn.edu

**Yi Ding** received his Ph.D degree from Beihang Univerity in 2014. From 2017, he began to work in the School of Information, Beijing WuZi University. His research interests concern blockchain technology, distributed system, high performance computing, cloud computing, big data technology, etc.
Email: dingyi@bwu.edu.cn

**Jian Zhu** received his BE in Sino-French Engineer School of Beihang University in 2019. He is now the master's degree student at Beihang University. His research interests include formal method, smart contract language.
Email: zhujian@buaa.edu.cn

**Jean-Pierre Talpin** did his PhD Thesis at Ecole des Mines de Paris under the advisory of Pierre Jouvelot, worked three years at the European Computer-Industry Research Centre in Munich and joined INRIA in 1995, in the EPART project-team of Paul Le Guernic. He led INRIA project-team ESPRESSO from 2000 to 2012 and now leads project-team TEA (on time in embedded architectures). He received the 2004 ACM SIGPLAN Award for the most influential POPL paper, with Mads Tofte; the 2012 ACM-IEEE LICS "Test of Time" Award, with Pierre Jouvelot; and the 2014 ITEA Award of Excellence.
Email: jean-pierre.talpin@inria.fr