Simulation Modelling Practice and Theory 51 (2015) 69–86

Contents lists available at ScienceDirect

Simulation Modelling Practice and Theory

journal homepage: www.elsevier.com/locate/simpat

Simulation of real-time systems with clock calculus

Kai Hu [a,*], Teng Zhang [b], Zhibin Yang [b,c,*], Wei-Tek Tsai [d]

[a] State Key Laboratory of Software Development Environment, Beihang University, Beijing, PR China
[b] School of Computer Science and Engineering, Beihang University, Beijing, PR China
[c] IRIT-CNRS, Université de Toulouse, Toulouse, France
[d] School of Computing, Informatics and Decision Systems Engineering, Arizona State University, Tempe, USA

ARTICLE INFO

Article history:
Received 12 July 2014
Received in revised form 20 October 2014
Accepted 23 October 2014
Available online 5 December 2014

Keywords:
SIGNAL
Clock calculus
Optimized clock tree
Code generation

ABSTRACT

Safety–critical real-time systems need to be modeled and simulated early in the development of lifecycle. SIGNAL is a data-flow synchronous language with clocks widely used in modeling of such systems. Due to the synchronous features of SIGNAL, clock calculus is essential in compilation and simulation. This paper proposes a new methodology for clock calculus that takes data dependencies into consideration. In this way, simulation code can be directly generated by using a depth-first traversal algorithm. In addition, a clock insertion method based on clock-implication checking is presented to obtain an optimized control structure.

1. Introduction

Safety–critical real-time systems such as automotive, avionics, and aerospace systems are usually reactive systems that need to interact with their environments. These systems often need to be modeled and simulated rigorously early in their development lifecycle.

Synchronous paradigm, based on *synchronous hypothesis* [1], is a formal method for modeling safety–critical systems. The synchronous hypothesis assumes that behaviors of the system will be represented as a sequence of discrete instants. At each instant, the system takes an input, performs computation, and produces an output with zero time. This abstraction simplifies timing analysis to allow the user to focus on the functional design without consideration of the computing platform. Other properties such as the temporal correctness can be considered when the platform has been chosen.

Synchronous languages such as Esterel [2], Lustre [3], QUARTZ [4] and SIGNAL [5], have been widely used. Esterel and QUARTZ are imperative languages while Lustre and SIGNAL are declarative. Apart from formal verification such as model checking and theorem proving, simulation is a major method for modeling of different kinds of systems [6].

An *abstract clock*, or a *clock*, is an important concept in synchronous languages. Each instant of a clock represents the status of computing including objects involved in computation. There are two kinds of clock models in synchronous languages:

* Corresponding authors. Tel.: +86 1082339460.
E-mail addresses: hukai@buaa.edu.cn (K. Hu), rahxephon89@163.com (T. Zhang), zhibin.yang@irit.fr (Z. Yang), wtsai50@gmail.com (W.-T. Tsai).

http://dx.doi.org/10.1016/j.simpat.2014.10.010
1569-190X/

- Mono-clocked model: In this model, there is a global clock and other clocks are derived from the global clock. This is used in Esterel, Lustre and QUARTZ; and
- Multi-clocked model: there is no global clock, and clock relations are defined relative to each other. This is used in SIGNAL.

In mono-clocked models, all behavior activations are controlled by the global clock, and events triggered in components are subsets of events triggered by the global clock. All the clocks of individual components must be synchronized with the global clock.

Safety–critical real-time systems, however, may contain components with individual clocks in a distributed environment. These clocks are not usually synchronized globally. Instead, components are loosely coupled so that each component can execute on its own rate and the synchronization is only needed among interacting components.

Based on a typical multi-clocked model called Polychronous paradigm [7], SIGNAL is widely used in the design of GALS (Globally Asynchronous Locally Synchronous) system. Furthermore, an IDE tool for modeling, analysis and simulation based on SIGNAL called Polychrony[1] has been developed. However, to correctly simulate the SIGNAL specification, generation of correct and efficient code is still a research topic. The first difficulty is to extract control structure from synchronous equations based on the clock model. While a multi-clocked model is more realistic, its compilation is also complicated as relations among clocks are far more complex. As a result, *clock calculus*, a process to resolve the clocks and construct the control structure of the executive code, is a principal method in compilation. Note that although having the same terminology "calculus", clock calculus, a formal method dedicated for the compilation of Synchronous language, takes a very different approach from *timed* $\pi - calculus$ [8] which is mainly for the modeling of real-time system. Many studies have been carried out on clock calculus. [9,10] proposed the methodology of the clock calculus used in Polychrony. The idea is to extract the system of Boolean equations from the program and develop a hierarchical relation among clocks. If the program is endochronous [7], a single-rooted clock tree can be constructed and the sequential code can be generated based on the clock tree. Others [11–13] focus on the resolution of numerical expressions to improve the compilation precision.

Additionally, not only clock relations, data dependency information is also needed to generated simulation code. However, the current methods tend to separate clock calculus from the data-dependency analysis. Two different structures are needed: one for the clock hierarchy and the other for data dependency. To avoid the situation that hierarchical clock relations may not be consistent with data dependencies(eg. clock-to-data cycle), the two structures need to be combined [14]. In addition, when being executed, code with hierarchical and nested control structure will be more efficient than the corresponding flat code because in the nest structure, guard conditions enclosed will not be checked unless the outer ones evaluate to true. Therefore, it is desirable to design a clock calculus to generate efficient control structure.

To address these problems, this paper proposes a new clock calculus to analyze SIGNAL programs and generate simulation code. The calculus considers data dependency in constructing the clock tree. By using the DFS (Depth-First Search) traversal algorithm, simulation code meeting the data dependency constraints can be generated. Moreover, by using the idea of clock-implication checking, a clock can be inserted as deep as possible to generate a hierarchical and nested control structure for the simulation code.

The paper is organized as follows: Section 2 introduces SIGNAL and its code generation process; Section 3 provides a methodology on the insertion of the tree, and deal with the data dependency when constructing the clock tree; Section 4 presents a case study to illustrate the process of clock calculus and the simulation of SIGNAL program; Section 5 presents the related works; and Section 6 concludes this paper.

2. Introduction to SIGNAL

SIGNAL program (also called model in the remainder of the paper) is usually a set of equations on which relations among signals or their clocks are induced. Thanks to the multi-clock feature, every signal has its own clock to specify at which instant it can carry a value. This section introduces SIGNAL, including its syntax and semantics.

2.1. Signals and clocks

Based on the synchronous hypothesis, behaviors of the system are described as discrete sequences of instants and the system does the input-computation-output at each instant. The value sequence $(x_t)_{t\in N}$ manipulated in SIGNAL is defined as a *signal* where t is an index to represent the position of instant. At each instant, a signal can be present or absent (denoted by $\perp$). When present, a signal can hold values typed with integer, real, complex or boolean. Apart from usual types, there is a special type called *event*. When an event signal is present, its value is true. Otherwise, it is absent.

The *clock* is a set of instants where a *signal* is present. The clock of signal s is denoted as $\hat{s}$, which is an event typed signal. Moreover, clock relations among signals are defined in the program either explicitly or implicitly.

[1] http://www.irisa.fr/espresso/Polychrony.

2.2. Primitive constructs

The program unit of SIGNAL is called *process* consisting of a set of equations specifying data and clock relations among signals. SIGNAL provides four primitive constructs: *instantaneous function*, *delay*, *down-sampling*, *deterministic merging*, to express such relations.

Instantaneous function: $y := f(x_1, \ldots, x_n)$, where $y, x_1, \ldots, x_n$ are signals, f is the n-ary operator, including arithmetical operator and Boolean operator. Signals are synchronous: (1) $x_1, \ldots, x_n$ have the same abstract clock; (2) when $x_1, \ldots, x_n$ are present, y is present and the value is evaluated to $f(x_1, \ldots, x_n)$; otherwise, y is absent. For instance, for function $x_3 := x_1 + x_2$, a execution trace is shown below.

x_1	1	$\perp$	4	...
x_2	2	$\perp$	5	...
x_3	3	$\perp$	9	...

Delay: $y := x \$ init\ c$, where y and x are signals and c is the initial value of y. The function of this operation is similar to a shift register: (1) at the first non-absent instant, the value of y is c; (2) at the following constants, whenever x is present, y will get the most recent value of x; (3) x and y have the same clock, meaning that y will be absent when x is absent. An execution trace is shown below.

x	1	$\perp$	3	2	2	...
y	c	$\perp$	1	3	2	...

Down-sampling: $y := x\ when\ b$, where y, x are signals and b is a signal or constant typed Boolean or event. The semantics is that when x is present and b is true, y is present and the value of y is the value of x. The implicit clock relation is $\hat{y} = \hat{x} \wedge [b]$ where $[b]$ means b is present and b is true. Here is a trace for this equation.

x	1	$\perp$	4	5	...
b	$\perp$	$\perp$	f	t	...
y	$\perp$	$\perp$	$\perp$	5	...

Deterministic merging: $y := x_1\ default\ x_2$, where y, x_1 and x_2 are signals. The semantics is that when x_1 is present, y is present and the value is x_1; when x_1 is absent and x_2 is present, y is present and the value is x_2; otherwise y is absent. Consequently, the clock relation is represented as $\hat{y} = \hat{x}_1 \vee \hat{x}_2$. The following is a trace for this equation.

x_1	1	$\perp$	$\perp$	5	...
x_2	2	$\perp$	3	$\perp$	...
y	1	$\perp$	3	5	...

Two other basic operators are *synchronous composition* and *local definition*.

Synchronous composition: $P = P1|P2$, where *P1* and *P2* are processes. The behavior of P is the conjunction of the mutual behaviors of *P1* and *P2*.

Local definition: *P1 = P where t_x1 x1; t_x2 x2; ...t_xn xn*, where *t_x1, ..., t_xn* are the types of signal *x1, ..., xn*. This conception is similar to the action scope of variables in programming languages. It means that signals defined within the process P will only be visible inside P.

From the introduction, one can give the abstract syntax process, shown as below.

$$P, Q :: = x := yfz|P|Q|P/x$$

The process (P and Q) consists of the synchronous composition ($P|Q$) of dataflow equations. P/x is the local definition of signals. Dataflow equation "$x := yfz$" represents that the value of x is decided by the input signal y, z and the operation f on them.

Apart from primitive constructs given above, SIGNAL also provides extended language such as unary *when*, memorization operator *cell* and operators explicitly specifying clock relations. However, all extended syntax can be rewritten in primitive constructs so that this paper assumes that the SIGNAL program is written in primitive constructs. For further introduction to SIGNAL syntax and semantics, reader can refer to [15,16].

2.3. System of Boolean equations and principles of simulation code

Compilation of SIGNAL program needs clock calculus to get the information about the relations among clock signals. As mentioned above, not only data dependencies, but also clock relations are implicitly defined in primitive constructs. Table 1 presents the clock relations for each primitive construct. In equations of instantaneous function and delay, signals are

Table 1
Primitive constructs and corresponding clock relations.

Primitive constructs	Clock relation
$y := f(x_1, x_2, \ldots, x_n)$	$\hat{y} = \hat{x}_1 = \hat{x}_2 = \cdots = \hat{x}_n$
$y := x\$ init\, c$	$\hat{y} = \hat{x}$
$y := x\, when\, z$	$\hat{y} = \hat{x} \wedge [z]$
$y := x\, default\, z$	$\hat{y} = \hat{x} \vee \hat{z}$

synchronous. These two operators are called mono-clocked operators. Down-sampling and deterministic merging, however, are multi-clocked operators because signals may have different clocks.

Equations of clock relations can be seen as Boolean equations, that, for each SIGNAL program, a system of Boolean equations can be extracted. Clock calculus deals with verification of the consistency among clocks and the generation of the control structure of the simulation code [17]. By resolving the system of Boolean equations, information of synchronization among signals or the inclusion relations among clocks can be obtained. If the program is endochronous, a single-rooted clock tree can be constructed. Each node in the tree is the clock and the parent-son relation is the inclusion relation between clocks: if n_1 is the father of n_2, clock of n_2 is included in clock of n_1.

Signal holds value whenever it is present. So the SIGNAL compiler uses the "**if** *guard* **then** *action*" as the control structure, in which *guard* is to check whether the clock is true and *action* is to get or compute the value of the signal.

Then a simple way for code generation is: (1) compute all clocks; (2) put all computation of signals into the right place according to the clock and data dependencies. However, to obtain more efficient code, nested condition is needed. A flatten control structure is shown below, in which action (c_1) is the action to be executed when clock c_1 is true.

```
c_1:=c_2 && c_3
if(c_2) then
  ...
if(c_1) then
  action(c_1)
```

It is easy to know that if c_1 = c_2 ∧ c_3, c_1 implies c_2 and c_3, which means that c_1 cannot be true if c_2 or c_3 is false. So we can get a nested control structure shown as below: only when c_2 is true, a check on whether c_3 is needed and it is obviously more efficient than the previous one. This paper will present a methodology to constructing clock trees to generate efficient executable code.

```
if(c_2) then
  if (c_3) then
    action (c_1)
```

2.4. Simulation framework for SIGNAL

Several studies have explored on how to use SIGNAL to model and simulate real-time systems. Polychrony, the framework for modeling, analysis and simulation based on SIGNAL, has been developed. The architecture is shown in Fig. 1. Apart from the use as a modeling language, SIGNAL can be also utilized as a synthesis formalism [18]. Architecture Analysis & Description Language(AADL) [19] is used to model the architecture of the system while Simulink[2] is used to describe the functional behaviors. Both models will be synthesised by SME [20], the metamodel of SIGNAL. APEX(for ARINC 653) and thread libraries written in SIGNAL [21] will be used during the modeling. Properties of the SIGNAL program such as deadlock freeness and reachability can be verified by model checking in SIGALI [22]. Functional behaviors of the system will be simulated by translating into executable code. [23] has proposed a translation to SynDex[3] for the distributed simulation. Moreover, software profiling can be carried out for the performance evaluation of SIGNAL specifications [24]. The SIGNAL simulation framework has been successfully used in the modeling and simulation of the door management system of Airbus A350 [25].

[2] http://www.mathworks.com/products/simulink/.
[3] http://www.syndex.org/index.htm.

3. Clock calculus and construction of clock tree

This section presents a new clock calculus and a strategy of the deep insertion and the way to maintain data dependency in the clock tree.

3.1. Process of clock calculus

3.1.1. Generating the 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 below:

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

In the definition, $\hat{x}$ is the clock of x; $\diamondsuit x$ represents the value of Boolean or event typed signal x. Note that in this paper, x, $\hat{x}$ and $\diamondsuit x$ will all be considered as *variables*. Furthermore, 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 2. Note that for the convenience of the resolution, there are at most two operands in the clock equations so that two auxiliary clocks variables, $\widehat{zt}$ and $\widehat{xdefault}$, are introduced.

3.1.2. Resolution of clock equations

The generated set of clock equations need 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. Compared with *SCE*, two more constraints are added to *SNF*:

- Each clock can be defined only once in *SNF*, which means in *SNF* there should not be more than one equation with the same left-hand side (LHS) value; and
- LHS values in *SNF* are not allowed to exist in right-hand side (RHS) of equations, this means clocks can only be defined by undefined clocks in *SNF*.

Algorithm *ClockToNF* 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 *eq* 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 recursive definition exists after the substitution, *eq2* will be put into *USNF* (lines 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* (lines 22–27). At the end of iteration, if *eq* has not been put into *USNF*, it will be put into *SNF* (line 28). This paper uses OBDD (Ordered Binary Decision Diagram) [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.

The time complexity of the algorithm is decided by several static and dynamic factors such as the size of *SCE*, the time consumed in the construction, substitution and comparison of the OBDD for clock equations and the size of *SNF* and *USNF* during the execution. Consequently, only some estimates can be given on this issue. All equations in *SCE* need to be represented by OBDD and the time complexity of the construction will be $O(2^n)$ where n is the number of variables in the boolean expression. However, since all equations in *SCE* will at most have two variables in the RHS expression, as shown in Table 2, the construction time can be seen as a constant time. The execution time of the OBDD substitution depends on the number of variables in the OBDD to be replaced and the size of set containing the definitions of these variables. For instance, the execution time of the statement in line 23 depends both on the number of variables in *eqd* and the number of elements in *SNF*. Furthermore, the number of executions of the loops in line 15 to line 20 or line 22 to line 27 have the same magnitude with 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 (as the check on equivalence of OBDDs is less complex than the substitution operation, the execution time in the loop is represented by w). This paper uses JDD (a Java implementation of BDD)[4]

[4] http://javaddlib.sourceforge.net/jdd/.

as the implementation method of OBDD. The website site shows that the performance is relatively good comparing with other implementations.

Algorithm 1. clockToNF

```
1: Inputs: SCE
2: Outputs: SNF
3: SNF ← ∅, USNF ← ∅
4: for all eq ∈ SCE do
5:   eq ← [SNF]eq
6:   if both sides of eq are complex expressions then
7:     USNF ← USNF ∪ {eq}
8:   else if eq.LHS is a complex expression ∧ eq.RHS is a variable then
9:     reverse(eq)
10:  end if
11:  if eq.LHS ∈ Vars(eq.RHS) ∨ eq.LHS has been defined in SNF then
12:    USNF ← USNF ∪ {eq}
13:  end if
14:  if eq ∉ USNF then
15:    for all eq2 is z = e ∈ SNF do
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:    TUSNF ← ∅
22:    for all eqd is e3 = e4 ∈ USNF do
23:      eqd ← [SNF]e3 = [SNF]e4
24:      if eqd.LHS and eqd.RHS are not equivalent then
25:        TUSNF ← TUSNF ∪ {eqd}
26:      end if
27:    end for
28:    USNF ← TUSNF, SNF ← SNF ∪ {eq}
29:    end if
30: end for
31: if NonEmpty(USNF)then
32:   error and exit the compilation
33: end if
34: return SNF
```

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

3.1.3. Generating clock equivalence classes and SRNF

After the resolution, unique definition for each clock (except for clocks on the *RHS* are undefined, usually clocks for input signals) is included in *SNF*. Some of them may have identical definitions, which means they are *synchronous*. Synchronous relation is *reflexive*, *symmetric* and *transitive* so that it is the equivalence relation on the set of clocks. To get more efficient code, the paper introduces the concept of *clock equivalence class* [27], which is a partition on the set of clocks $X = \{X_i | i \in Z^+\}$. For each X_i, its elements $\hat{c}_1, \ldots, \hat{c}_k$ are clock variables which are synchronous with each other. Definition of 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.

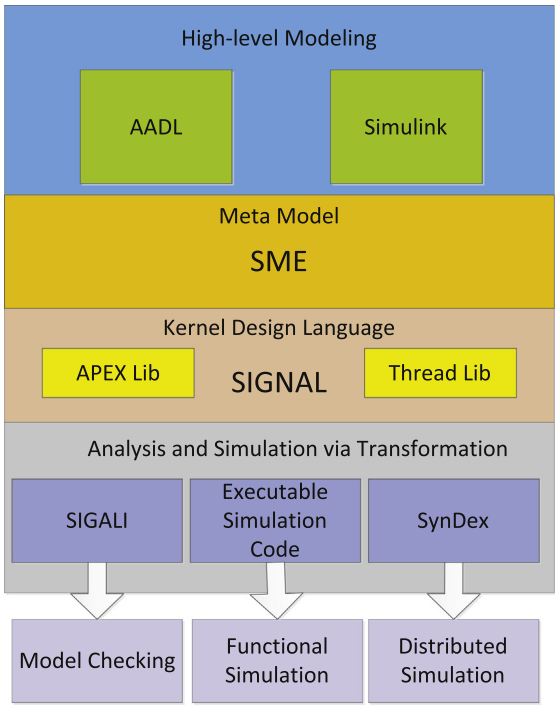

Fig. 1. Modeling, analysis and simulation framework for SIGNAL.

Table 2
Primitive constructs and corresponding clock equations.

Primitive constructs	Clock relation
$y := f(x_1, x_2, \ldots, x_n)$	$\hat{y} = \hat{x}_n, \hat{x}_1 = \hat{x}_n, \ldots, \hat{x}_{n-1} = \widehat{x_n}$ (if f returns a Boolean value, add the equation $\Diamond x := f(\Diamond x_1, \ldots, \Diamond x_n)$)
$y := x\$ init\, c$	$\hat{y} = \hat{x}$
$y := x when z$	$\widehat{zt} = \widehat{z} \wedge \Diamond z, \hat{y} = \hat{x} \wedge \widehat{zt}$
$y := x default z$	$\hat{y} = \hat{x} \vee \widehat{z}, x\widehat{default} = \widehat{z} \setminus \hat{x}$

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*, which can be obtained by replacing clocks with their class in *SCE*. The BNF definition is shown below:

NCE::=ClassId=e
$e ::= ClassId | \Diamond x_1 | Cond | \neg e | e \vee e | e \wedge e | e \setminus e$

ClasseId is the *LHS* value of the equation and *RHS* is the expression specifying relations on classes. $\Diamond x$ 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.

3.2. Clock tree definition, properties and tree construction

By previous steps, clock equivalence classes and relations among them have been obtained in *SRNF*, from which hierarchical relations among clocks can be extracted. [27] gives the definition of clock hierarchy relation "$\leqslant$":

- for Boolean signal x, there are relations $\hat{x} \leqslant [x]$ and $\hat{x} \leqslant [\neg x]$;
- for variables b and c, if there are relations $b \leqslant c$ and $c \leqslant b$, then b and c are synchronous; and
- for clock equation $b_1 = c_1 \diamondsuit c_2, \diamondsuit \in \{\wedge, \vee, \backslash\}$, if there are relations $b_2 \leqslant c_1, b_2 \leqslant c_2$, then there exists relation $b_2 \leqslant b_1$.

If relation $b \leqslant c$ exists, c is determined or extracted from b, this means that only when b is true, c can be evaluated to be true. The corresponding implication relation is represented as $c \rightarrow b$. If the program is endochronous, the hierarchy is the single-rooted tree called the *clock tree*, the control structure of the simulation code. [9] proposes a process to generate clock trees. This paper proposes a new methodology with the following features:

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

Here, the definition of the clock tree and properties are first given below.

3.2.1. Definition and properties of the clock tree

[9] introduces a clock tree. Clock tree is a two-tuple $< V, f >$ where

- V is the set of nodes, and
- $f : V \rightarrow V$ is the function defined on V, satisfying the condition: there is a node $r \in V$, for any node $v \in V$, there is a positive integer n and $r = f^n(v), f(r) = r$. r is the unique root of the tree.

Here we give definitions of some useful terminologies.

Parent node, direct-son node: for $x, y \in V$, if $y = f(x)$, then y is the parent node of x and x is the direct-son node of y. In the clock tree, a node can only have one parent node and the parent node of the root node is itself.
Ancestor node, descendant node: for $x, y \in V$, if $y = f^n(x)$ where n is a positive integer, y is the ancestor node of x and x is the descendant node of y.
Brother node: for $x, y \in V$, if $f(x) = f(y)$, x, y are the brother nodes of each other. If x is on the left side of y, x is the left-brother of y, denoted as $x \in LB(y)$, and y is the right-brother of x, denoted as $y \in RB(x)$.

For convenience, this paper gives another form of definition for the clock tree.

Definition 2. A clock tree is a two-tuple $< VS, \leqslant >$ where

- VS is the set of nodes of a clock tree(denote as clock node) and
- $\leqslant$represents the binary relation between two clock nodes, that if $v1 \leqslant v2$, $v1$ is the parent node of $v2$.

Definition 3. A clock node is defined as a four-tuple $< Class, RNF, F, AS >$ where

- *Class* represents the clock equivalence class corresponding to this node;
- *RNF* is the reduce normal form equation defining *Class*;
- *F* is the parent node; and
- *AS* is the ordered list of *signal definition equations* (also denoted as *assignments*) representing the actions to be executed when the clock of the node is evaluated to true

To generate the correct executable code from a SIGNAL program, data dependency relations need to be met by the clock tree obtained:

- for nodes x and y in the tree, if y is the ancestor of x, *LHS* value of assignments in x cannot be the *RHS* values of assignments in y; and
- for nodes x and y in the tree, if $x \in LB(y)$, *LHS* value of assignments in y cannot be the *RHS* value of assignments in x.

If the properties are satisfied, the data dependency relations will be maintained by the DFS traversing of the tree from left to the right. Note that it is assumed that data dependencies within the node are kept so that only the data dependencies among nodes need to be considered.

Table 3
Signal definition equations for primitive constructs and input signals.

Syntax	Signal definition equations	Clock equivalence class
$y := f(x_1, x_2, \ldots, x_n)$	$y = f(x_1, x_2, \ldots, x_n)$	$SCEC[OBDD(y)]$
$y := x\,when\,z$	$y = x$	$SCEC[OBDD(y)]$
$y := x\,default\,z$	$y = x$	$SCEC[OBDD(x)]$
$y := x\,default\,z$	$y = z$	$SCEC[OBDD(z \setminus x)]$
$y := x\$init\,c$	*NULL*	*NULL*
$?type\_xx$	$read(x)$	$SCEC[OBDD(x)]$

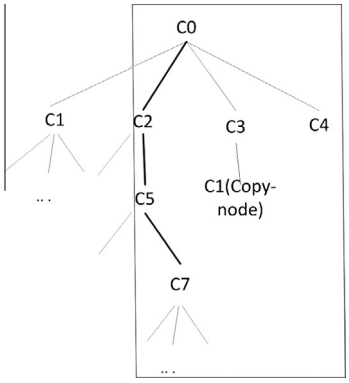

Fig. 2. A clock tree example.

3.2.2. Construction of clock tree

When constructing a clock tree, properties mentioned above need to be preserved. This can be done with three steps.

- Divide all *signal definition equations* into corresponding clock equivalence classes;
- Sort the signal definition equations and clock definition 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 definition equations and attached to the corresponding clock equivalence class. The mapping relation is shown in Table 3. Note that *SCEC[OBDD(y)]* means the clock equivalence class of signal *y*. Moreover, since there is no data dependency relation in *delay*, corresponding equation does not exist.

After this process, each class *C* in *SCEC* has a set of assignments(*C.Eq*). Along with *clock definition equations* in *SRNF*, these equations will be sorted to obtain the ordered list *Elist* with the following properties:

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

The clock tree is constructed by traversing the *Elist* and equations in the *Elist* will be attached to the clock node. Assume that for equation $x = e$, the node it is attached to is denoted as *N*, then for any node *M* containing operands in expression *e*, one of the relations listed below exists:

- $M = f^n(N)$, n is the positive integer;
- $f(N) = f(M)$, and $M \in LB(N)$; and
- there exists a positive integer m, $P = f^m(N)$, and $M \in LB(P)$.

The relations mentioned above ensure the data dependency relations are preserved in the tree. Algorithm *treeConstruction* illustrates the process of the traversal. If the current equation (denoted as *eq*) is a clock definition equation (line 6), since no node representing the clock has been inserted in the tree, a new node called *Vc* is created (line 9). *Vf* is the node to which Vc will be attached (lines 9–10). If the current equation is a signal definition equation (line 13), first it is needed to find a proper node in the tree to attach (lines 13–14). If no node is found, a so-called *copy-node* is created (line 17). As its name indicates,

there has (have) been a node (or nodes) representing the same clock in the tree but the equation is not allowed to attach to it (anyone of them) due to the violation of properties mentioned above. The *copy-node* will be inserted (lines 18–19) and *eq* will be then attached to it (line 21).

Algorithm 2. treeConstruction

```
1: Inputs: Elist
2: Outputs: VS
3: Vr ← VS ∪ Vr
4: for all eq ∈ Elist do
5:   if eq is the clock definition equation then
6:     RST ← genPath(eq.RHS)
7:     Vf ← findInsertClock(RST, eq.LHS)
8:     Vc ← createNode(eq.LHS), Vc.RNF ← eq
9:     buildRelation(Vf, Vc)
10:    VS ← VS ∪ Vc
11:  else
12:    find corresponding clock equivalence class of eq, denoted as C
13:    RST ← genPath(eq.RHS ∪ C)
14:    Vc ← findInsertData(RST, C)
15:    if Vc = NULL then
16:      Vf ← findInsertClock(RST, C)
17:      Vc ← createNode(C)
18:      buildRelation(Vf, Vc)
19:      Vs ← VS ∪ Vc
20:    end if
21:    Vc.AS ← Vc.AS ∪ eq
22:  end if
23: end for
24: return VS
```

To preserve the clock and data dependency relations, a limit branch called the *dependency path* of the node needs to be created first(by *genPath()* at line 6 and line 13). Dependency path is an ordered list of nodes in the tree $n_1, \ldots, n_k$, in which n_1 is the leaf, n_k is the root and for any neighbor nodes $n_x, n_y, y = x + 1, n_y = f(n_x)$. Assume *NS* is the set of nodes which contains all definitions of the clock or signal on which the target node depends. The node n_1 is the rightmost and deepest node in the tree belonging to *NS*. An example of the clock tree is shown in Fig. 2. In the figure, C7 is the leaf node. From it, a path $C0 \rightarrow C2 \rightarrow C5 \rightarrow C7$ can be obtained. The target node needs to be inserted at the right side of the path(the rectangle part, denoted as *RST*). Similarly, when attaching signal definition equation, the node to be attached needs to hold the same condition. If no node is found in *RST*, the copy-node will be created(lines 13–14).

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, shown in Algorithm *findInsertClock*.

Algorithm 3. findInsertClock

```
1: Inputs: Elist, SCEC
2: Outputs: VS                                   ▷ clock tree
3: Vr ← createNode(R)                            ▷ create clock node for the root clock
4: while nonEmpty(Queue) do                      ▷ Queue is not empty
5:   head ← deQueue(Queue)                       ▷ get the head from Queue
6:   for all node ∈ directchildrenofhead do
7:     if node ∈ RST ∧ CN.Class → node.Class then ▷ node is in RST and the implication relation holds
8:       Queue ← addNode(node, Queue)            ▷ put node into Queue
9:     end if
10:  end for
11: end while
12: return head
```

A queue called *Queue* is used for the traversal. Note that the direct children of the node is an ordered list: the node inserted later will be at the right side of the nodes inserted earlier. The root will be first put into Queue, then the iteration begins: get the head of the node denoted as *head*; for each element node belonging to the direct children of head, if it is at the right side of the path ($node \in RST$) and the clock of *CN* implies clock of node ($CN.Clock \rightarrow node.Clock$), put node into the rear of Queue. The iteration will not stop until Queue is empty. *CN* will be inserted as the direct child of the last element got from Queue.

The time complexity of algorithm *treeConstruction* is decided by the size of *Elist* (denoted as n where number of signal definition equations is p) and number of clock equivalence classes in *SCEC* (denoted as m). At the worst case, the number of nodes in the tree will be $p + m$ (every single definition equation corresponds to a clock node in the tree). As the time complexity of BFS and DFS in a tree are both $O(V)$ where V is the number of nodes in the tree, the magnitude of the time complexities of *genPath*, *findInsertClock* and *findInsertNode* are $O(p + m)$. Consequently, the worst performance of *treeConstruction* is $O(n * (p + m))$.

3.3. Generation of simulation code

The sequential simulation code can be generated only when the SIGNAL program is endochronous: the scheduling of the computation can be obtained in the compilation, independent from the execution environment. There is only one master clock in the clock hierarchy and all other clocks can be extracted from it. The clock tree can be used as the control structure. Furthermore, as data dependency relations have been preserved, sequential code can be easily generated by the Depth-First Search of the clock tree.

In the clock tree, the hierarchy relations indicate the implication among clocks: if node n is the descendant of node m, the code generated from n will be nested in code generated from m. For the order of code among brothers, node on the left side will be generated prior to the right side. For each node, the code block takes the form "**if** *C* **then** *AS*" where *C* is the test to the clock and AS is the set of assignments to be executed. The algorithm DFSGen is shown below, and it traverses a clock tree recursively from left to right and from up to down. If multiple nodes have the same clock, the clock will be defined once only.

Algorithm 4. DFSGen

```
1: Inputs: node                                  ▷ clock node of the tree
2: if isRoot(node) == false then                 ▷ if node is not the root node
3:    generateClockDef(node.RNF)                 ▷ generate the clock definition
4:    generateI f(node.Class)                    ▷ generate the guard condition of the clock
5: end if
6: for all assignment ∈ node.AS do               ▷ generate signal definition equations in the node
7:    generateAssign(assignment)
8: end for
9: for all N ∈ node.F^-1 do                      ▷ node.F^-1 is the list of direct children of node
10:   DFSGen(N)                                  ▷ recursively generate code for these nodes from left to right
11: end for
```

4. Case study

Fig. 3 illustrates overall simulation process. The first step is to model the system, then the simulation code is generated by compilation. After simulation execution, the result will be analyzed to provide guidance to modify the model. This section will illustrate the proposed clock calculus to generate simulation code.

ABRO is a common example for illustration of synchronous programming [28–30]. It is a data collection process used by a system to collect data from two channels *A* and *B*. The finite state machine of *ABRO* is shown in Fig. 4. ABRO has three input signals: *A*, *B* and *R* and one output signal *O*. Symbol "?" and "!" respectively denote the receipt and emission of signal. At the initial state, ABRO waits for the input signal. When both A and B are received at the same time or by any order, O will be outputted. Signal R plays the reset role of the system:

(a) Once O is outputted, ABRO will not receive the input from neither A or B until it receives R and return to the initial state;
(b) If R is received after one of A or B has been received, ABRO will return to the initial state, waiting for the next inputs of A and B.

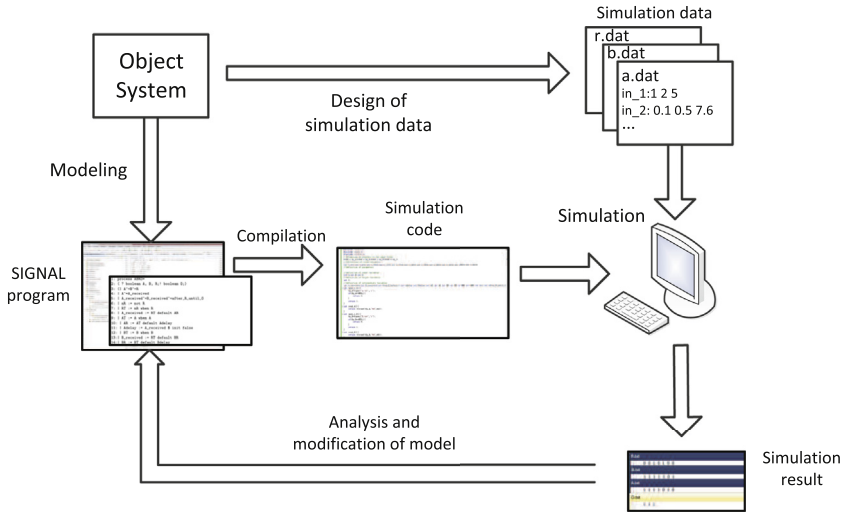

Fig. 3. Process of SIGNAL simulation.

SIGNAL program of *ABRO* is shown in Fig. 5.

(1) **Generating clock equations**
For each dataflow equation, a corresponding clock equation is generated to extract the clock relation. For the *ABRO* process, dataflow equations from line 3 to line 11 and the corresponding clock equations generated are shown in Table 4. All clock equations will be put into set *SCE*.

(2) **Resolution of the clock equations**
The generated set of clock equations will be resolved to check if the program is consistent and obtain the unique definition for the clock of signals. For instance, clock equations shown in Table 4 will be resolved as shown in Fig. 6. One can see that every signal has its own unique and flatten definition while the only undefined signal is $\hat{}R$.

(3) **Generating clock equivalence class and Reduced Normal Form**
Clocks will be divided into clock equivalence classes and then *Reduced Normal Form* can be generated to represent the relations among classes. *SCEC* and *SRNF* generated for the example program are shown in Figs. 7 and 8. One can see that all clocks of signals have been divided into corresponding clock equivalence classes and the only undefined clock is *C2* so the program is endochronous.

(4) **Constructing the clock tree**
After generating *signal definition equations* from dataflow equations, *signal definition equations* and *clock definition equations* will be sorted according to the data and clock dependency relations. Fig. 9 shows part of the result of the sort.
By traversing the *Elist*, the clock tree is constructed. Fig. 10 illustrates the structure of the clock tree. Every node contains the clock definition and ordered list of signal definition equations. Clock node of *C_13* is shown in Fig. 11. One can observe that *C_2* is the father of *C_13*; the clock definition equation for *C_13* is $C\_13 = C\_2 \wedge A$; two signal definition equations are attached to the node and they have been sorted according to the data dependencies.

(5) **Generating code**
Following the Polychorny compiler [14], this paper uses the iteration style to generate the sequential code. Each step of iteration simulates one instant: reading the input, computing and writing the output. The core part of the iteration is shown in Fig. 12. According to the definition of clock tree and order of the traversal, the generation order of the code is $C\_2, C\_13, C\_77, C\_6, C\_75, C\_26, C\_79, C\_82, C\_84, C\_86$ and C_92. One can observe that lines 2–8 are actions under C_2; line 11 and 12 are actions triggered by C_13. Furthermore, since C_84 implies C_82 and C_86 implies C_84, the definition of C_84 and C_86 are respectively put under the trigger of C_82 and C_84.

(6) **Simulating the SIGNAL program**
A compiler has been implemented to validate the generation of sequential code. After the SIGNAL compilation, the simulation code is compiled and executed in Visual Studio 2010 to analyze the behavior of ABRO. An execution trace is shown in Table 5. As *A*, *B* and *R* are synchronous and belong to the root clock, value of each signal will be read at each step of iteration. Here, value 1 corresponds to true while value 0 correponds to false. For instance, at instant *t1*, A and B are both evaluated to value 1 and R is evaluated to value 0 so that O is outputted as value 1. At instant *t2*, however, O is absent because R has not be received to reset the state. According to the specification of ABRO and inputs, *O* will be

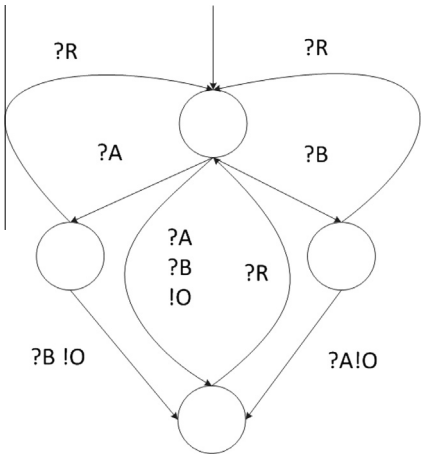

Fig. 4. Finite state machine of ABRO process [28].

```
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 R
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;
```

Fig. 5. Signal program of ABRO process.

Table 4
Dataflow equations and corresponding clock equations.

Dataflow equations	Clock equations
$A^{\wedge}=B^{\wedge}=R, A^{\wedge}=A\_received,$	$\widehat{A}=\widehat{R}, \widehat{B}=\widehat{R}, \widehat{A\_received}=\widehat{R}$
$A\_received^{\wedge}=B\_received^{\wedge}=after\_R\_until\_O$	$\widehat{B\_received}=\widehat{R}, \widehat{after\_R\_until\_O}=\widehat{R}$
$nR := \ not\ R$	$\widehat{nR}=\widehat{R}, nR = not\ R$
$RT := nR\ when\ R$	$\widehat{RT}=\widehat{nR}\wedge\widehat{R\_true}, \widehat{R\_true}=\widehat{R}\wedge\Diamond R$
$A\_received := RT\ default\ AR$	$\widehat{A\_received}=\widehat{RT}\vee\widehat{AR}, \widehat{A\_received\_default}=\widehat{AR}\setminus\widehat{RT}$
$AT := A\ when\ A$	$\widehat{AT}=\widehat{A}\wedge\widehat{A\_true}, \widehat{A\_true}=\widehat{A}\wedge\Diamond A$
$AR := AT\ default\ Adelay$	$\widehat{AR}=\widehat{AT}\vee\widehat{Adelay}, \widehat{AR\_default}=\widehat{Adelay}\setminus\widehat{AT}$
$Adelay := A\_received\ \$init\ false$	$\widehat{Adelay}=\widehat{A\_received}$

$$
\begin{aligned}
&1:\ \widehat{A\_received} = \widehat{R}\\
&2:\ \widehat{B\_received} = \widehat{R}\\
&3:\ \widehat{after\_R\_until\_O} = \widehat{R}\\
&4:\ \widehat{A} = \widehat{R}\\
&5:\ \widehat{B} = \widehat{R}\\
&6:\ \widehat{nR} = \widehat{R}\\
&7:\ nR = not\ R\\
&8:\ \widehat{RT} = \widehat{nR}\ \&\&\ \diamond R\\
&9:\ \widehat{A\_received\_default} = \widehat{R} \backslash (\widehat{R}\ \&\&\ \diamond R)\\
&10:\widehat{A\_true} = \widehat{R}\ \&\&\ \diamond A\\
&11:\widehat{AT} = \widehat{R}\ \&\&\ \diamond A\\
&12:\widehat{AR} = \widehat{R}\\
&13:\widehat{AR\_default} = \widehat{R} \setminus (\widehat{R}\ \&\&\ \diamond A)\\
&14:\widehat{Adelay} = \widehat{R}
\end{aligned}
$$

Fig. 6. *SNF* extracted from clock equations in Table 4.

$$
\begin{aligned}
&C\_2 = \{\widehat{R}, \widehat{A\_received}, \widehat{B\_received}, \widehat{after\_R\_until\_O}, \widehat{A}, \widehat{B}, \widehat{R}, \widehat{nR}, \widehat{RR}\}\\
&, \widehat{Adelay}, \widehat{Bdelay}, \widehat{from\_R\_before}\\
&C\_6 = \{\widehat{RT}, \widehat{R\_true}, \widehat{Re}\}\\
&C\_75 = \{\widehat{A\_received\_default}, \widehat{B\_received\_default}, \widehat{RR\_default}\}\\
&C\_13 = \{\widehat{A\_true}, \widehat{AT}\}\\
&C\_26 = \{\widehat{B\_true}, \widehat{BT}\}\\
&C\_77 = \{\widehat{AR\_default}\}\\
&C\_79 = \{\widehat{BR\_default}\}\\
&C\_84 = \{\widehat{ABR}, \widehat{Arr\_true}\}\\
&C\_92 = \{\widehat{from\_R\_before\_O\_default}\}\\
&C\_82 = \{\widehat{after\_R\_until\_O\_true}, \widehat{Arr}\}\\
&C\_86 = \{\widehat{ABR\_true}, \widehat{nO}, \widehat{O}\}
\end{aligned}
$$

Fig. 7. Clock equivalence classes of the example program.

$$
\begin{aligned}
&C\_6 = C\_2\ \&\&\ \diamond R\\
&C\_75 = C\_2 \setminus C\_6\\
&C\_13 = C\_2\ \&\&\ \diamond A\\
&C\_77 = C\_2 \setminus C\_13\\
&C\_26 = C\_2\ \&\&\ B\\
&C\_79 = C\_2 \setminus C\_26\\
&C\_82 = C\_2\ \&\&\ \diamond after\_R\_until\_O\\
&C\_84 = C\_82\ \&\&\ Arr\\
&C\_86 = C\_84\ \&\&\ ABR\\
&C\_92 = C\_2 \setminus C\_86
\end{aligned}
$$

Fig. 8. Part of SRNF of the example program.

$$
\begin{aligned}
&1:read(A)\\
&2:read(B)\\
&3:read(R)\\
&4:C\_13 = C\_2\ \&\&\ \diamond A\\
&5:C\_77 = C\_2 \setminus C\_13\\
&6:C\_6 = C\_2\ \&\&\ \diamond R\\
&7:C\_75 = C\_2 \setminus C\_6\\
&8:C\_26 = C\_2\ \&\&\ \diamond B\\
&9:C\_79 = C\_2 \setminus C\_26\\
&10:C\_82 = C\_2\ \&\&\ after\_R\_until\_O\\
&11:nR = not\ R\\
&12:AT = A\\
&14:AR = Adelay\\
&15:BT = B\\
&\ldots
\end{aligned}
$$

Fig. 9. Part of *Elist* of the example program.

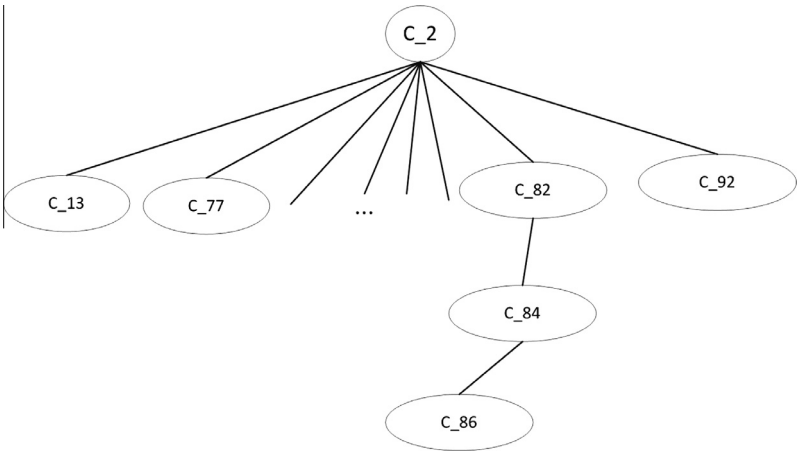

Fig. 10. Clock tree of the example program.

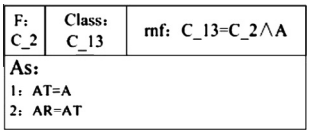

Fig. 11. Clock node of *C*_13.

present and evaluated to value 1 at instant *t1*, *t4* and *t7*. After executing the simulation code, *VCD (Value Change Dump)* file will be generated and illustrated in GTKWave,[5] shown in Fig. 13. In the figure, 1 ps corresponds to a instant and red rectangle represents absent. From top to bottom, four waves depict the trace of A,B,R and O. From the wave, one can observe that O is absent at 1 ps, 3 ps, 5 ps and 6 ps, consistent with expect result shown in Table 5 so that the generated C code is able to simulate the behavior of the *ABRO* program correctly.

(7) **Evaluation of the simulation code**

Implication checking is used in this paper to generate nested control structure. As shown in Fig. 12, if clock *C_82* is evaluated to false, code from line 21 to line 31 (denoted as *BlockC_82*) will not be executed. According to the specification of ABRO process, C_82 is true whenever signal *after_R_until_O* is true. Assume that the maximum execution time of core iteration function is *T_core*, the maximum execution time of BlockC_82 is *t_b* and the probability of C_82 evaluated to true is *p*. As a result, the average execution time will be $T\_average = T\_core * p + (T\_core - t\_b) * (1 - p)$ and the time saved is $T\_core - T\_average = t\_b * (1 - p)$, denoted as *t_s*. Note that in any execution trace, the value of *p* is 0.5 at most as signal *O* cannot be 1 at two successive instants so that the maximum average time saved at each instant will be $t\_b * 0.5$.

An experiment has been done to evaluate the execution time of the generated simulation code. The experiment is done on a PC with a CPU of Intel(R) Core(TM)2 Quad Q9400 2.66 GHz and a 4 GB memory. The OS is Windows 7 and the simulation code is compiled and executed in Visual Studio 2010. The paper uses the win32 API *QueryPerformanceFrequency* and *QueryPerformanceCounter* to count the time elapsed during the code execution. Table 6 shows the data of the experiment. The value of *p* is $1254/1387 \doteq 0.9$ and $t\_s = (1 - 0.9) * 1.53 = 0.153us$. The overall time saved is $(1387 - 1254) * t\_s = 20.35us$. The experiment result illustrates that the nested control structure can effectively decrease the execution time of the simulation code.

5. Related work

The compilation of SIGNAL has been studied before. [9,10] present various clock calculi. First, a system of Boolean equations is constructed according to the clock relations to specify the calculation order of the clocks. Then, partition trees are constructed from the Boolean-valued signal. In each tree, clock of the child node is included in the clock of the parent node. After this procedure, some clocks are divided into the partition trees while others are the one node tree. Finally, by fusing all clock trees, hierarchical structure of the clock is built. During the fusion, triangularity is preserved in the depth-first search. The generated clock tree is used as the control structure of the sequential code. However, no information of the data dependency is specified in the tree. [14] proposes HCDG(Hierarchical Conditional Dependence Graph) which is composed

[5] http://www.gtkwave.sourceforge.net.

```
1:int core() {
2: if(read_A()==EOF)
3:      return 0;
4: if(read_B()==EOF)
5:      return 0;
6: if(read_R()==EOF)
7:      return 0;
8: nR=!R;
9: C_13=C_2&&A;
10: if(C_13!=0){
11:      AT=A;
12:      AR=AT;
13: }
14: C_77=C_2&&  !C_13;
15: if(C_77!=0){
16:      AR=Adelay;
17: }
...
18: C_82 = C_2 && after_R_until_O;
19: if(C_82!=0){
20:      Arr=B_received;
21:      C_84=C_82&&Arr;
22:      if(C_84!=0){
23:           ABR=A_received;
24:           C_86=C_84&&ABR;
25:           if(C_86!=0){
26:                O=1;
27:                write_O();
28:                nO=!O;
29:                from_R_before_O=nO;
30:           }
31:      }
32: }
33: C_92=C_2&&!C_86;
34: if(C_92!=0){
35:      from_R_before_O=RR;
36: }
37: return 1;
38:}
```

Fig. 12. Core part of the iteration.

Table 5
An execution trace of the program.

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	$\perp$	$\perp$	1	$\perp$	$\perp$	1

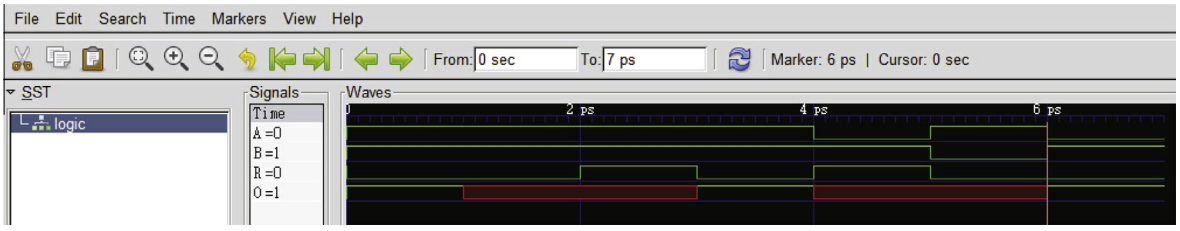

Fig. 13. Wave diagram of the simulation result.

of a clock hierarchy and a conditioned scheduling graph to generate sequential code. The clock hierarchy and scheduling graph will be generated separately and then combined together, making the compilation more complicated. As for the research on the optimization of clock tree, [9] presents that the code generation can take advantage of the inclusion relation in the clock tree but it is too trivial.

Another problem of the clock calculus is to deal with numerical expressions. At the current implementation of the compiler, the numerical expressions are seen as the black boxes that may lose information useful for the checking of synchronization relations among signals. [11] uses the methodology of abstract interpretation and IDD(Interval-Decision Diagram) to deal with the numerical expressions. Translation from SIGNAL program to its interval abstraction and to IDD is defined so that the numerical properties can be verified. [12] proposed a non-intrusive methodology for the enhancement of the compilation of SIGNAL based on the combined numerical-Boolean abstraction. By using the SMT-solver, abstraction is reasoned to find the empty clocks, mutual clocks and the inconsistency among clock relations. [13] proposed a language called Clock Language(CL) that is more expressive than the Clock Algebra used in the clock calculus. CL is based on an extension by numerical aspects of the purely Boolean clocks used in the SIGNAL context. SAT solver is used to prove the clock

Table 6
Data of the experiment.

T_core	*t_b*	Execution times of core function	Execution times of BlockC_82
46.1us	1.53us	1387	1254

properties. [31] presents a verified transformation from polychronous specification to a variant of *Clock Guarded Actions* called S-CGA. The semantics preservation is proved using Coq, a proof assistant based on a higher-order logic [32]. The study is also the foundation of the work proposed in this paper, from which the correctness of the clock calculus can be verified in the same idea.

Furthermore, some studies focuses on the distributed or multi-threaded code generation for SIGNAL programs. [14] proposes two methods, static scheduling and dynamic scheduling, to generate multi-threaded code. In static scheduling, compiler will generate clusters based on the scheduling graph. Every cluster has its own clock tree. In dynamic scheduling, every dataflow equations corresponds to a micro-thread. Each thread synchronizes with other threads using atomic action "wait-notify" according to the data dependencies. But these methods requires target SIGNAL program has to be endochronous, which means there has to be only one root clock but this property is too strict and unnecessary in the multi-thread code generation. [33] defines property called weak endochrony which allows the program to have multiple root clocks. The concurrency can be then exploited to generate distributed or multi-threaded code. Based on weak endochrony theory, [34,35] respectively presents methods to generate multi-threaded code. [36] proposes a methodology generating parallel OpenMP code from SIGNAL program but clock calculus is not discussed. [37] presents a method translating S-CGA [31] to multi-threaded code and maps multi-thread code to multi-core architecture.

Compared with existing approaches, this paper makes the following contributions:

- Different from the method proposed in [14], this paper combines clock hierarchy with data dependencies to reduce the compilation complexity(this is done in Section 3). Moreover, evaluations on major algorithms have shown that the time complexities are acceptable for practical use.
- Clock-implication checking is used in the construction of the clock tree so that the clock can be inserted as deep as possible to make the simulation code efficient(this is done in Section 3.2.2). Specifically, an experiment has been done in Section 4 showing that the nest control structure can effectively reduce the execution time of the simulation code.
- The proposed new clock calculus method is used to generate simulation code(this is done in Section 4). By executing the simulation code, the behavior of synchronous programs can be verified.

6. Conclusion

This paper proposed a new clock calculus for the simulation of the SIGNAL program. By resolving the system of Boolean equations, relations among clocks can be obtained to generate the clock tree. When developing the clock tree, clock-implication checking is used for the deeper insertion. Deeper the node is inserted, higher the efficiency of the control structure will be. Furthermore, data dependencies are considered during the insertion of the tree. By simply using depth-first search, executive code satisfying both clock and data dependency relations can be generated. The time complexity of major algorithms have also been carried out to show the practicality of the method. Moreover, the whole process has been illustrated by using an example. The performance of the generated code illustrates that the method has improved the efficiency of the simulation code.

In the future, we will take the resolution of numerical expression into consideration. In addition, to generate code for correctly simulating the behavior of the SIGNAL program, the process of the compilation needs to be verified. The formalization and verification of the methodology is one of our future work.

Acknowledgment

This work was supported by National Natural Science Foundations of China (No. 61073013), State Key Laboratory of Software Development Environment (No. SKLSDE-2014ZX-09) and Aviation Science Foundation of China (No. 2012ZC51025). Grateful acknowledgment is made to Mr. Mamoun FILALI-AMINE and Prof. Jean-Paul BODEVEIX from IRIT-CNRS. They have given a lot of instructive advice to this paper.

References

[1] D. Potop-Butucaru, R. de Simone, J.-P. Talpin, The synchronous hypothesis and synchronous languages, in: The Embedded Systems Handbook, 2005, pp. 1–21.
[2] G. Berry, G. Gonthier, The esterel synchronous programming language: design, semantics, implementation, Sci. Comput. Programm. 19 (2) (1992) 87–152.
[3] N. Halbwachs, P. Caspi, P. Raymond, D. Pilaud, The synchronous data flow programming language lustre, Proc. IEEE 79 (9) (1991) 1305–1320.
[4] K. Schneider, The synchronous programming language quartz, Tech. rep., Internal Report 375, Department of Computer Science, University of Kaiserslautern, Kaiserslautern, Germany, 2009.
[5] P. LeGuernic, T. Gautier, M. Le Borgne, C. Le Maire, Programming real-time applications with signal, Proc. IEEE 79 (9) (1991) 1321–1336.

[6] W. Tsai, X. Sun, Q. Huang, H. Karatza, An ontology-based collaborative service-oriented simulation framework with microsoft robotics studio, Simul. Modell. Pract. Theory 16 (9) (2008) 1392–1414.
[7] P. Le Guernic, J.-P. Talpin, J.-C. Le Lann, Polychrony for system design, J. Circ. Syst. Comput. 12 (03) (2003) 261–303.
[8] N. Saeedloei, G. Gupta, Timed π-calculus, in: Trustworthy Global Computing – 8th International Symposium, TGC 2013, Buenos Aires, Argentina, August 30-31, 2013, Revised Selected Papers, 2013, pp. 119–135. doi:http://dx.doi.org/10.1007/978-3-319-05119-2_8.
[9] T.P. Amagbegnon, L. Besnard, P. Le Guernic, et al., Arborescent canonical form of boolean expressions, Tech. Rep. 2290, INRIA, 1994.
[10] P. Amagbégnon, L. Besnard, P. Le Guernic, Implementation of the data-flow synchronous language signal, ACM SIGPLAN Notices 30 (6) (1995) 163–173.
[11] 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>.
[12] 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>.
[13] M. Nebut, Specification and analysis of synchronous reactions, Formal Aspects Comput. 16 (3) (2004) 263–291.
[14] 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>.
[15] L. Besnard, T. Gautier, P. Le Guernic, Signal v4-inria version: Reference manual, IRISA, March 2004.
[16] Z. Yang, J.-P. Bodeveix, M. Filali, A comparative study of two formal semantics of the signal language, Front. Comput. Sci. 7 (5) (2013) 673–693.
[17] M. Nebut, An overview of the signal clock calculus, Electron. Notes Theor. Comput. Sci. 88 (2004) 39–54.
[18] H. Yu, Y. Ma, T. Gautier, L. Besnard, P.L. Guernic, J.-P. Talpin, Polychronous modeling, analysis, verification and simulation for timed software architectures, J. Syst. Architect. 59 (10, Part D) (2013) 1157–1170.
[19] SAE, Architecture Analysis & Design Language (AADL) v2, AS-5506, SAE International, 2004.
[20] C. Brunette, J.-P. Talpin, A. Gamatié, T. Gautier, A metamodel for the design of polychronous systems, J. Logic Algebraic Programm. 78 (4) (2009) 233–259. {IFIP} WG1.8 Workshop on Applying Concurrency Research in Industry.
[21] A. Gamatié, T. Gautier, Modeling of avionics applications and performance evaluation techniques using the synchronous language signal, in: Proceedings of SLAP03, ENTCS, vol. 88, Elsevier, 2003.
[22] H. Marchand, P. Bournai, M.L. Borgne, P.L. Guernic, Synthesis of discrete-event controllers based on the signal environment, Discr. Event Dynam. Syst. 10 (4) (2000) 325–346.
[23] H. Yu, Y. Ma, T. Gautier, L. Besnard, J.-P. Talpin, P.L. Guernic, Y. Sorel, Exploring system architectures in aadl via polychrony and syndex, Front. Comput. Sci. 7 (5) (2013) 627–649.
[24] A. Kountouris, P.L. Guernic, Profiling of signal programs and its application in the timing evaluation of design implementations, in: The IEEColloq. on HW-SW Cosynthesis for Reconfigurable Systems, HP Labs, 1996.
[25] H. Yu, Y. Ma, Y. Glouche, J.-P. Talpin, L. Besnard, T. Gautier, P.L. Guernic, A. Toom, O. Laurent, System-level co-simulation of integrated avionics using polychrony, in: SAC, 2011, pp. 354–359.
[26] S.B. Akers, Binary decision diagrams, IEEE Trans. Comput. 100 (6) (1978) 509–516.
[27] 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.
[28] A. Gamatié, Designing Embedded Systems with the SIGNAL Programming Language, Springer, 2010. p. 29.
[29] G. Berry, The foundations of esterel, in: Proof, Language, and Interaction, 2000, pp. 425–454.
[30] G. Berry, The Esterel v5 language primer: version v5_91, Centre de mathématiques appliquées, Ecole des mines and INRIA, 2000.
[31] Z. Yang, J.-P. Bodeveix, M. Filali, K. Hu, D. Ma, A verified transformation: from polychronous programs to a variant of clocked guarded actions, in: International Workshop on Software and Compilers for Embedded Systems (SCOPES 2014), ACM, 2014, pp. 128–137.
[32] B. Barras, S. Boutin, C. Cornes, et al., The Coq Proof Assistanct Reference Manual Version 6.1, INRIA, May 1997.
[33] D. Potop-Butucaru, B. Caillaud, A. Benveniste, Concurrency in synchronous systems, Formal Methods Syst. Des. 28 (2) (2006) 111–130.
[34] B.A. Jose, S.K. Shukla, H.D. Patel, J. Talpin, On the deterministic multi-threaded software synthesis from polychronous specifications, in: 6th ACM/IEEE International Conference on Formal Methods and Models for Co-Design, 2008 (MEMOCODE 2008), IEEE, 2008, pp. 129–138.
[35] D. Potop-Butucaru, Y. Sorel, R. de Simone, J.-P. Talpin, From concurrent multi-clock programs to deterministic asynchronous implementations, Fundam. Inform. 108 (1) (2011) 91–118.
[36] K. Hu, T. Zhang, Z. Yang, Multi-threaded code generation from signal program to openmp, Front. Comput. Sci. 7 (5) (2013) 617–626.
[37] Z. Yang, J. Bodeveix, M. Filali, Multi-core code generation from polychronous programs with time-predictable properties, in: Proceedings of the First International Workshop on Architecture Centric Virtual Integration co-located with the 17th International Conference on Model Driven Engineering Languages and Systems, ACVI@MoDELS 2014, Valencia, Spain, September 29, 2014, 2014. <http://ceur-ws.org/Vol-1233/acvi14_submission_4.pdf>.