DOI 10.1007/s11704-013-3906-4

Multi-threaded code generation from Signal program to OpenMP

Kai HU (✉)[1], Teng ZHANG[2], Zhibin YANG[2,3]

1 State Key Laboratory of Software Development Environment, Beihang University, Beijing 100191, China
2 School of Computer Science and Engineering, Beihang University, Beijing 100191, China
3 IRIT-CNRS, Université de Toulouse, Toulouse 31062, France

Abstract The use of multi-core processors will become a trend in safety critical systems. For safe execution of multi-threaded code, automatic code generation from formal specification is a desirable method. Signal, a synchronous language dedicated for the functional description of safety critical systems, provides soundness semantics for deterministic concurrency. Although sequential code generation of Signal has been implemented in Polychrony compiler, deterministic multi-threaded code generation strategy is still far from mature. Moreover, existing code generation methods use certain multi-thread library, which limits the cross platform executions. OpenMP is an application program interface (API) standard for parallel programming, supported by several mainstream compilers from different platforms. This paper presents a methodology translating Signal program to OpenMP-based multi-threaded C code. First, the intermediate representation of the core syntax of Signal using synchronous guarded actions is defined. Then, according to the compositional semantics of Signal equations, the Signal program is synthesized to dependency graph (DG). After parallel tasks are extracted from dependency graph, the Signal program can be finally translated into OpenMP-based C code which can be executed on multiple platforms.

Keywords multi-thread, synchronous language, Signal, code generation, OpenMP

E-mail: hukai@buaa.edu.cn

1 Introduction

Multi-core processors have been widely used in high-performance computing and universal computing. With the increase of functional and non-functional demands, multi-core architecture will become indispensable in safety critical systems such as avionics, aerospace and automobile control.

Multi-threaded software is necessary to make full use of computing resources of multi-core processors. At present, two types of strategies have been developed to aid programming multi-threaded software. One is application program interfaces (APIs) and libraries provided by Unix-like OS [1] and Windows [2]; the other includes several parallel programming technologies such as MPI [3] for the multi-processor distributed system, and OpenMP [4] and Intel TBB [5] for the shared memory architecture which provides mechanism to describe high level parallel algorithm.

However, these two strategies fail to satisfy the strict quantitative indicators of functional and non-functional properties demanded in safety critical systems. If the executions of the embedded software are non-deterministic, they may cause undesirable consequences such as delay of reactions and race conditions. In addition, parallel programming is error-prone because programmers have to specify the synchronization and resources sharing among threads. This will bring the multi-threaded coding for safety-critical applications a high-risk programming activity [6].

To solve this problem, using model-based development and automatic code generation technology based on formal methods has become a trend in academics and industries. One of the available formal methods used in safety critical systems is the synchronous language [7, 8], built on a mathematical model combining synchronous hypothesis and deterministic concurrency. In synchronous hypothesis, time is abstracted as partial discrete logical time series and actions executed by the system are abstracted as discrete steps of computing. The input, computation and the output take no time at each instant (the unit of discrete logical time). Due to the abstract time model, the inherent functional properties are preserved, which makes synchronous languages suitable for the functional design of systems. The mainstream synchronous languages include Esterel [9], Lustre [10], and Signal [11] among which Signal is a multi-clocked language that no global clock is pre-defined and every signal has its own clock. Compared to the mono-clocked synchronous languages, multi-clocked model is more suitable for the description of distributed systems and multi-core systems.

Endochrony [12] and weak endochrony [13] properties have been proposed to generate deterministic code from Signal. In the endochronous Signal program, the clock of each signal can be computed from a "root clock". The Polychrony compiler [14, 15] not only supports the sequential code generation, but also provides the function of multi-threaded code generation from the endochronous program, based on the clustering method. According to the data dependency relations, the Signal program is divided into tasks which will be "forked" as threads at the runtime. These threads will communicate with each other by the "wait-notify" system call while in each thread the sequential code will be executed. However, there are still some implicit concurrencies in the program which may not be discovered and the use of "wait-notify" takes time in the synchronization among threads.

Weak endochrony property, as its name implies, is less strict than endochrony. If the relation among signals meets the full-diamond condition [13], it is possible to generate deterministic multi-threaded code. [16] proposes a methodology checking weak endochrony property based on bounded model-checking. Since the model-checking method is expensive for the code generation, the paper proposes another method based on the isochrony [17] property which is suitable for the compositional design. This method, however, cannot fully cover all the weakly endochronous programs. [18] proposes a methodology generating deterministic multi-threaded code from weakly endochronous program based on synchronous flow dependence graphs [19]. Every statement in the program corresponds to a thread and the threads synchronize with each other based on "wait-notify" system call. On the basis of the atom theory proposed in [13,20] presents a general method to check weak endochrony on multi-clocked synchronous programs. The corresponding strategy of multi-threaded code generation is given in [21]. However, some restrictions must be met. For instance, data types of the program interface should be finite and delay equations should be replaced by the clock relation equations. Therefore, some weakly endochronous programs may be rejected.

Another strain of methodology is proposed in [22]. It translates synchronous guarded actions, an intermediate representation for mono-clocked synchronous languages, into OpenMP-based multi-threaded program. The synchronous guarded actions are first translated into dependency graph (DG) and then, from the DG, tasks are divided for the parallel execution. Finally, OpenMP-based C code can be generated according to the task partition. Since OpenMP has been implemented by several compilers from different OS, the generated code can be executed on multiple platforms.

As an API standard for parallel programming, OpenMP provides abundant mechanism for the description of high level parallel algorithms. The newest version of OpenMP supports fine-grained scheduling and task balancing which can increase the performance of the program. However, few studies of multi-threaded code generation for Signal have chosen OpenMP as the target language. Drawing on the idea presented in [22], this paper introduces a methodology discovering the implicit parallelism from the Signal program and translating the endochronous Signal program to OpenMP-based C code. However, some vital changes are made to fit the characteristics of Signal. Firstly, while [22] directly translates the program written by synchronous guarded actions, primitive constructs in Signal are needed to be first translated into representation of synchronous guarded actions. With regard to this, some new features are added to synchronous guarded actions. For instance, in order to represent implicit clock relations defined in each primitive construct, Boolean variables representing the clocks are introduced. Moreover, in [22], action dependency grpah (ADG) is a bipartite graph in which variables and guarded actions are vertices. This paper proposes a DAG (directed acyclic graph)-like form of DG in which nodes are guarded actions and edges represent the dependency relations between nodes.

The paper is structured as follows. An informal introduction to the Signal is provided in Section 2. In Section 3, the paper proposes an intermediate representation of the core syntax of Signal using synchronous guarded actions. Based on the compositional semantics and data dependency relations, the formal definition of DG is given. In Section 4, methods of finding the implicit parallelism of the Signal program and the task partition from DG are proposed. Finally, in Section 5, the translation from partitioned tasks to OpenMP-based C code is defined and an example is analyzed to validate the methodology proposed in this paper.

2 Introduction to Signal

2.1 Syntax and corresponding semantics

As mentioned in Section 1, time is abstracted and the behaviors of the system are divided into a discrete series of instants. At each instant, the input, computing and output are executed instantaneously and simultaneously. The unbounded series of typed values are called *signals*. Signals in the program can be present or absent at each instant and the *clock* of a signal is defined as the series of subscripts at which instant the signal is present.

In Signal, primitive constructs (core syntax) are provided to express the relations between signals, defined in Table 1. Note that the clock of signal s is denoted as "ˆs".

Operators in Signal not only depict the data dependency relations but also imply clock relations among signals. According to the clock relations, operators can be divided into two types, mono-clocked operators and multi-clocked operators. In equations of mono-clocked operators, including Relation and Delay, operand signals are synchronous, that is, at any instant, all signals will be present or absent at the same time. In contrast, operand signals of multi-clocked operators, such as Sampling and Merge, may have different clocks. For instance, in the Sampling equation, shown in Table 1, the left-hand side value O will be present only when the right-hand side value $s1$ and $s2$ are present and $s2$ evaluates to true.

Table 1 Primitive constructs of Signal

Name	Syntax	Informal semantics
Relation	$O := f(s1, s2, \ldots, sn)$	When $s1$, $s2$, $\ldots$, sn are present, O is present and the value is $f(s1, s2, \ldots, sn)$; otherwise signal O is absent
Delay	$O := s1$ \$ init c	The clocks of O and $s1$ are equal; when $s1$ is present, the value of O is the previous present value of $s1$; the initial value of O is c
Sampling	$O := s1$ when $s2$	O will be present and evaluated to $s1$ only when $s1$ and $s2$ are present and $s2$ evaluates to true
Merge	$O := s1$ default $s2$	When $s1$ is present, O is present and evaluated to $s1$; otherwise when $s2$ is present, O is present and evaluated to $s2$; if neither $s1$ nor $s2$ is present, signal O is absent

Apart from primitive constructs listed in Table 1, Signal also provides other extended constructs such as the clock operator "∧" and memory operator "cell". Moreover, nested process, module and other mechanisms are defined in Signal to specify the large system with components at various rates. The details of the syntax can be referred in [23].

Relations among signals and their clocks are defined as *equations* in Signal. The basic unit of a Signal program, called *process*, consists of a set of equations. Two basic operators, respectively called *synchronous composition* and *local definition*, are applied to the process. The syntax and corresponding semantics are shown in Table 2.

Table 2 Primitive operations on the process

Name	Syntax	Informal semantics
Synchronous composition	$P \vert Q$	P and Q are processes. The behavior of $P \vert Q$ is the conjunction of the mutual behaviors of P and Q [24]
Local definition	P where t_1 $s1$ $\ldots$ t_n sn ; end	P is a process and $s1$ $\ldots$ sn are signals. The scope of $s1, \ldots, sn$ is restricted to P which means they are not visible outside P [24]

From the introduction given above, we 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.

To simplify the translation, this paper sets a few restrictions on the source Signal program: all equations are written in primitive constructs; every signal can only be defined once in the program [24]; all equations are in the same process, which means that there is no subprocess in the Signal program. Moreover, in Signal, a program is a process and shares the same syntax [24]. In the remainder of the paper, the source program is a flattened process written in primitive constructs.

Two IDEs, RT-builder [25] and Polychrony [26] are based on Signal. The former one is commercial version and Polychrony is open source for academic use. The code generated by Polychrony compiler takes the form of the infinite loop of elementary iterations. In each iteration, the program will read from the input, compute and write to the output. More details of this code generation principle can be found in [14]. In this paper, we will use the iteration as the execution model of the generated code.

2.2 An example of Signal

A Signal program presented in [24] called ABRO is used to illustrate the code generation in the paper. Figure 1 is a finite state machine specification of the ABRO process.

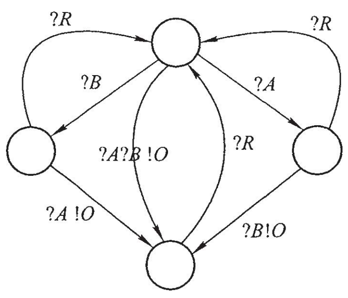


Fig. 1 A finite state machine specification of the ABRO process

ABRO emits signal O when input signals A and B have been received. When input signal R is received, ABRO comes back to the initial state and begins to wait the inputs. Once O has been emitted, it will not be emitted again until R has arrived to reset the state. The original version of the Signal program of ABRO can be found in [24]. Here we give a modified version, shown in Fig. 2.

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

Fig. 2 Signal program of ABRO process

In the process ABRO, A, B, R are Boolean typed input signals and O is event typed output signal, as shown in Line 2. Line 3 to Line 20 are dataflow equations specifying the clock and value relations among signals. Lines 3 and 4 synchronize the input signals A, B, R with the intermediate signals A_received, B_received and after_R_until_O. By analyzing the clock relations among signals, it can be deduced that the clock of input signals is the only root clock of this program, so the program is endochronous.

3 Intermediate representation for primitive constructs and the program synthesis method

Due to the declarative feature of Signal, one of the indispensable steps when generating imperative code is to translate the source program into an intermediate form with the information of clock hierarchy and data dependency relations. Section 3.1 presents a method translating primitive constructs of Signal into the code block of synchronous guarded actions which is used to represent the clock and data dependencies among the operands of equations. To represent the data dependency relations for the whole program, another intermediate form called DG is defined in Section 3.2. Implicit concurrency of the program can be then detected by analyzing the DG, which will be proposed in Section 4. Note that the source program to be translated should be endochronous. Based on the definition given in [18], the informal description of endochrony is: a Signal program is endochronous if and only if the clock of all signals can be computed according to the internal clock relations and no external environment runtime information is needed, which equivalently means that there will be a root clock in the program.

3.1 Synchronous guarded actions for primitive constructs

Based on the semantics of core syntax given in Section 2, the synchronous guarded actions is defined as below, which is different from [22]:

A synchronous guarded action is a four-tuple $\langle R, L, B, O\rangle$. R is the set of signals which represent the right-hand side values of the primitive constructs. L is the set of the left-hand side values. B is the code block defined as $\langle G, A\rangle$, taking the form "if G then A". G is a Boolean expression and A is the set of actions to be executed when G holds. O is the output set of the signals which can be used in other blocks' right-hand side values.

There are three kinds of signals: input, output and intermediate signals. Although input signals cannot be the left-hand side value of the equation, the clock relations can be specified to decide at which instants the value can be read. Intermediate signals and output signals can be the left-hand and right-hand side value of the equation.

The synchronous guarded actions representations of the core syntax are defined below. We use syntax of C as the style of pseudo code in the block and the clock of signal "s" is denoted as "C_s". Note that before constructing blocks for each equation in the program, clock analysis needs to be completed to divide all signals into clock equivalence classes so that synchronous signals will have the same clock representation in each block.

a) $O := f(s1, s2, \ldots, sn)$

```
Right-hand side signals: $s1, s2, \ldots, sn, C_O$
Left-hand side signal: $O$
if($C_O$==true)
   $O = f(s1, s2, \ldots, sn)$;
Output of the block: $O$
```

In the block above, "f" is an n-ary instant operator. The right-hand side operands of the block are the operands of operator "f" and the left-hand side operand of the block is O. The implicit clock relation is "$\hat{}O = \hat{}s1 = \cdots = \hat{}sn$" while C_O is defined as the common clock of these signals. Signals on the right-hand side are needed to compute the value of O. The Boolean expression in the block, "C_O==true", means that the O can be computed only when O is present at this instant. The Output of the block depicts that after the execution of the code block, O can be used as a right-hand side value and if O is an output signal, the write action can be executed.

b) $O := s1$ default $s2$

```
Right-hand side signals: $C\_s1, s1$
Left-hand side signals: $O, C\_O$
if($C\_s1$==true){
   $O = s1$;
   $C_O$=true;

Output of the block: $C\_O, O$
```

```
Right-hand side signals: $C\_s1, C\_s2, s2$
Left-hand side signals: $O, C\_O$
if($C\_s1$==false && $C\_s2$==true){
   $O = s2$;
   $C\_O$=true;

Output of the block: $C\_O, O$
```

From the semantics of operator Merge, two corresponding code blocks are constructed. Signal $s1$ is prior to $s2$. If $s1$ is present, O is assigned to the value of $s1$. If $s1$ is absent and $s2$ is present, O is assigned to the value of $s2$. Note that apart from the assignment to O, the clock of O, denoted by C_O should be assigned to true if $s1$ or $s2$ is present.

c) $O := s1$ when $s2$

```
Right-hand side signals: $s2$, $C\_s1$, $C\_s2$
Left-hand side signals: $O, C\_O$
if($C\_s1$==true && $C\_s2$==true && $s2$==true){
   $O = s1$; $C\_O$=true;

Output of the block: $C\_O, O$
```

From the block shown above, we can see that if $s1$ and $s2$ are present and $s2$ evaluates to true (which means type of $s2$ should be Boolean or event), O is present and evaluates to the value of $s1$.

d) $O := s1$ \$ init c

```
Right-hand side signals: $C\_O$
Left-hand side signals: Null
if($C\_O$==true){
}
Output of the block: Null
```

For operator Delay, since Delay is a mono-clocked operator, $s1$ and O have the same clock, which means that when $s1$ is present at the instant, O is also present and can be used as right-hand side value. The clock of O and $s1$, denoted as C_O, is the single right-hand side signal. However, the corresponding code block has no action since no data dependency is defined in the equation. How to assign value to the memory signal will be introduced in Section 3.2.

As for the input signals in the root clock set, at the beginning of each iteration, read actions should be executed. The corresponding clock should also be set to true.

```
Right-hand side signals: Null
Left-hand side signals: $i$
{read($i$); $C\_i$=true;}
Output of the block: $i$, $C\_i$
```

For the input signal not belonging to the root clock set, the clock can be extracted from the clock calculation, denoted as C_s. If C_s evaluates to true, the read actions can be executed so that the read will be nested in the same block assigning C_s to true.

```
Left-hand side signals: $i$, other signals
if(...){
   other assignments
   C_s=true;
   read($i$);

Output of the block: $i$, other signals
```

Two other primitive constructs include the local definition (P/x) and synchronous composition ($P|Q$). They have no corresponding code blocks of synchronous guarded actions. The local definition operator enables one to restrict the scope of a signal to a process [24]. Intermediate signals are defined in this part and they are invisible from the outside of the process. Synchronous composition is the union of equations defined in the program. Equations communicate with each other by common signal variables. The behavior of the program can be seen as the conjunction of mutual behaviors of all equations [24]. Based on the semantics of synchronous composition, code blocks of synchronous guarded actions generated from the program will be composed into the DG according to the data dependencies among code blocks of synchronous guarded actions, used to describe the behavior of the whole program and explore the implicit concurrencies.

3.2 Synthesis method based on DG

After generating the code block of synchronous guarded actions for each equation in the program, DG can be constructed. All signals belonging to the same class are synchronous. The definition of DG is given below: DG is defined as $\langle NS, \rightarrow\rangle$. NS is the set of nodes which represents the code block of synchronous guarded actions. $\rightarrow$ is the precedence relation between nodes defined over NS as follow: $s1 \rightarrow s2$ if and only if some signals exist both in right-hand side of $s2$ and left-hand side of $s1$, which indicates that to execute the code in $s2$, we first need to get the execution result of $s1$. Note that two code blocks for the operator Merge defined in Section 3.1 will be treated as one node in DG. A DG is correct if every cycle "$s0 \rightarrow \cdots \rightarrow sn \rightarrow s0$" is a pseudo cycle: the conjunction of all guard expression of synchronous guarded action $s0, \ldots, sn$ involved in the cycle is false. From this, it is also easy to know that DG is not strictly a DAG because there may be pseudo cycle in the graph. Note that if dependencies from clocks to their corresponding signals (input signals) are added, clock constraints are set. In this case, values read from the environment must meet the constraints to guarantee the correct execution.

To generate a complete DG, some situations need to be considered. Some blocks are for the read of input signal with no right-hand side signal. These nodes will be composed into a single node called the initial node. Since there is no precedence among these nodes, they can be arranged at any order when getting composed. Furthermore, the clock of the root clock class, denoted as C_1, has to be set to true in the front of the initial node. At the end of iteration, a terminal node is also needed in which every signal on the left-hand side of the Delay equation(denoted as memory signals) will be set to new value under the condition that it has been present at the last iteration. These blocks are then synthesized into the terminal node in which all the clocks of the clock equivalence classes are also needed to be set to false. Note that the initial assignments to memory signals needs to be executed before the iteration begins. Furthermore, according to the definition given in Section 3.1, some blocks (denoted as $b1$) may have the right-hand side signals which are memory signals but the code block of synchronous guarded actions (denoted as $b2$) for Delay equation has no left-hand side value. In this case, tests on whether clocks of memory signals evaluate to true are included in the guard condition of $b1$ and no precedence relation needs to be specified between nodes respectively containing $b1$ and $b2$.

Redundancies in the generated DG can be found as follows:

1) The Boolean sub expressions may be duplicated in the condition expression of the code blocks. For instance, according to the algorithm given in Section 3.1, the corresponding condition of the equation "$c := b$ when not b" is "*C_b*==true&& *C_b*==true &&*b*==false".

2) If there are blocks with signals in the same clock equivalence class as the left-hand side value, there will be duplicated assignments to the clock.

To eliminate these redundancies, the duplicated Boolean expressions are to be deleted first. Then traverse from the initial node, if there is a clock assignment in a node, the same assignment in its subsequent nodes will be deleted.

4 Task partition strategy

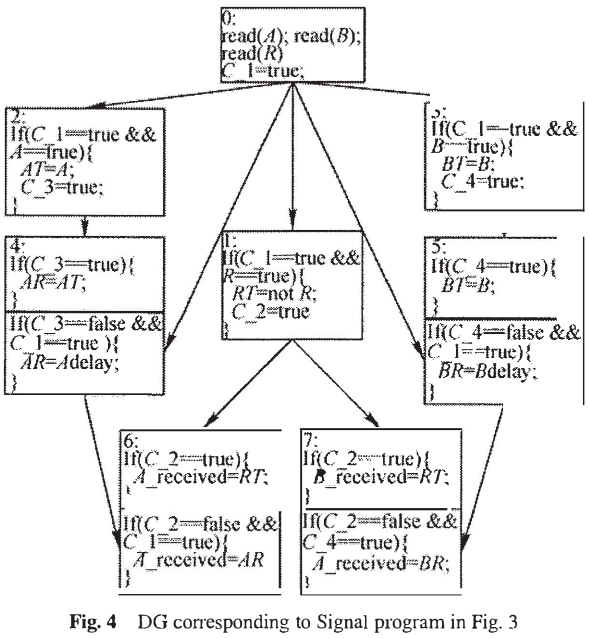
As defined in Section 3, DG is a kind of DAG on which precedence relation is defined among nodes. Informally, the precedence relation indicates the dependency between nodes. If no precedence relation exists between two nodes, they can run in parallel. In this section, we partition a set of nodes of DG into tasks. The precedence relation on tasks is compatible with the precedence relation on nodes.

Task is defined as a set of nodes belonging to NS of DG. TS is a two-tuple $\langle T, \twoheadrightarrow\rangle$, in which T is a partition of NS and $\twoheadrightarrow$ is the precedence relation among tasks on T. A task t of T is an anti-chain in the reflexive transitive closure $\rightsquigarrow$ of $\rightarrow$ (i.e., nodes in a task cannot be compared: $(\forall t \in T)(\forall n1 \in t)(\forall n2 \in NS)(n1 \rightsquigarrow n2) \Rightarrow ((n1 = n2) \vee (n2 \notin t))$. Among tasks belonging to T, $t1 \twoheadrightarrow t2$ if and only if there exists at least one node in $t2$ which is preceded by nodes in $t1$. Note that although cycle checking has been done after the construction of DG, there may be pseudo cycles of tasks since pseudo cycles are allowed in DG. To deal with this problem, the guard of a task t is defined as the disjunction of the guards of nodes belonging to t. If the conjunction of all these guards of tasks involved in the cycle is false, the cycle is a pseudo cycle. Because of pseudo cycles, the result of some nodes in $t2$ may be required by some nodes in $t1$ (when some condition C is true) and conversely (when the condition C is false). In this paper we only consider DAG of nodes: the processing of programs with cycles and pseudo cycles is not described here. One can then use a topological sorting to partition nodes into tasks so that the result of the partition is a total order of tasks: for all tasks $t_1, t_2, \ldots, t_n$ belonging to T, a series of them, $t_1 \twoheadrightarrow \cdots \twoheadrightarrow t_{m_{n-1}} \twoheadrightarrow t_{m_n}$ exists. As a result, no task pair will be allowed to execute in parallel. Here we illustrate the task partition with the example in Section 2.

Part of the code is shown in Fig. 3. Lines 3 and 4 show the root clock of the program is the clock of input signals *A*, *B* and *R*. Synchronized with these signals, intermediate signal *A*_received, *B*_received and after_*R*_until_*O* also belong to the root class. The initial node, as a result, contains the read of the input signals and the assignment to the root clock *C*_1. Then for each equation in the program, corresponding code block of synchronous guarded actions are generated. Finally, these nodes are composed into the DG shown in Fig. 4. Arrows in the figure represent the precedence relation "→". Note that since *Adelay* and *Bdelay* are memory signals and the guard expression "*C*_1==true" implies that *A*delay and *B*delay are present (*A*delay and *B*delay are in the root clock

```
3:  (| A_received ^= B_received ^= after_R_until_O
4:   | A_received ^= A ^= B ^= R
5:   | RT := not R when R
6:   | A_received := RT default AR
7:   | AT := A when A
8:   | AR := AT default Adelay
9:   | Adelay := A_received $ init false
10:  | BT := B when B
11:  | B_received := RT default BR
12:  | BR := BT default Bdelay
13:  | Bdelay := B_received $ init false
```

Fig. 3 Fragment of ABRO process



```
0:
read(A); read(B);
read(R)
C_1=true;
```

```
2:
If(C_1==true &&
A==true){
  AT=A;
  C_3=true;
}
```

```
3:
If(C_1==true &&
B==true){
  BT=B;
  C_4=true;
}
```

```
4:
If(C_3==true){
  AR=AT;
}
If(C_3==false &&
C_1==true ){
  AR=Adelay;
}
```

```
1:
If(C_1==true &&
R==true){
  RT=not R;
  C_2=true
}
```

```
5:
If(C_4==true){
  BT=B;
}
If(C_4==false &&
C_1==true){
  BR=Bdelay;
}
```

```
6:
If(C_2==true){
  A_received=RT;
}
If(C_2==false &&
C_1==true){
  A_received=AR
}
```

```
7:
If(C_2==true){
  B_received=RT;
}
If(C_2==false &&
C_4==true){
  A_received=BR;
}
```

Fig. 4 DG corresponding to Signal program in Fig. 3

class), corresponding nodes for two Delay equations are omitted and there is no arrow explicitly illustrating the precedence relation between *A*delay and *AR* nor between *B*delay and *BR*.

The result of the task partition is shown in Fig. 5. Nodes of DG are divided into four tasks. Arrows in the figure illustrate the total order among four tasks: task1 will be executed first and task4 will be the last one to be executed. Nodes in the same task can be executed in parallel. For instance, in task2 there are three nodes preceded by the node in task1. However, there is no precedence relations among these nodes so that they can be executed in parallel.

After the task partition, the generated tasks will be used for the OpenMP code generation which will be presented in Section 5.

5 OpenMP based C code generation and case study

OpenMP, an API for shared-memory parallel programming in C/C++ and FORTRAN, provides users with several mechanisms such as compiler directive, programming interface and environment variables for the high level description of parallel algorithms. This section will introduce the method mapping tasks partitioned in Section 4 to the OpenMP-based C code.

The basic syntax of directives in OpenMP is shown in Fig. 6. There are several directives in OpenMP. For instance, directive "parallel for" is used for the parallelization of "for" loop; directive "parallel sections" is used to specify the code blocks which can be executed in parallel. In OpenMP 3.0, directive "task" is added to support the parallelization of irregular data, iteration and recursive call. Since actions of code blocks are simple computations, we choose the directive "parallel sections" for the parallelization, shown in Fig. 7. The code blocks executed in parallel are encircled in directive "#pragma omp section" respectively.

Moreover, race condition will occur when multiple threads can access shared variables at the same time, which will make the result of the execution non-deterministic. Clauses, such as private, shared and reduction, are provided to specify the variable scope and sharing property to handle this problem. Clause private (list) is used to declare that each thread has its own duplicate of the variables in the list. Clause shared (list) declares the list of shared variables among threads. Clause reduction (operator:list) specifies an operation on one or a list of variables. Each thread has duplicates of variables in the list

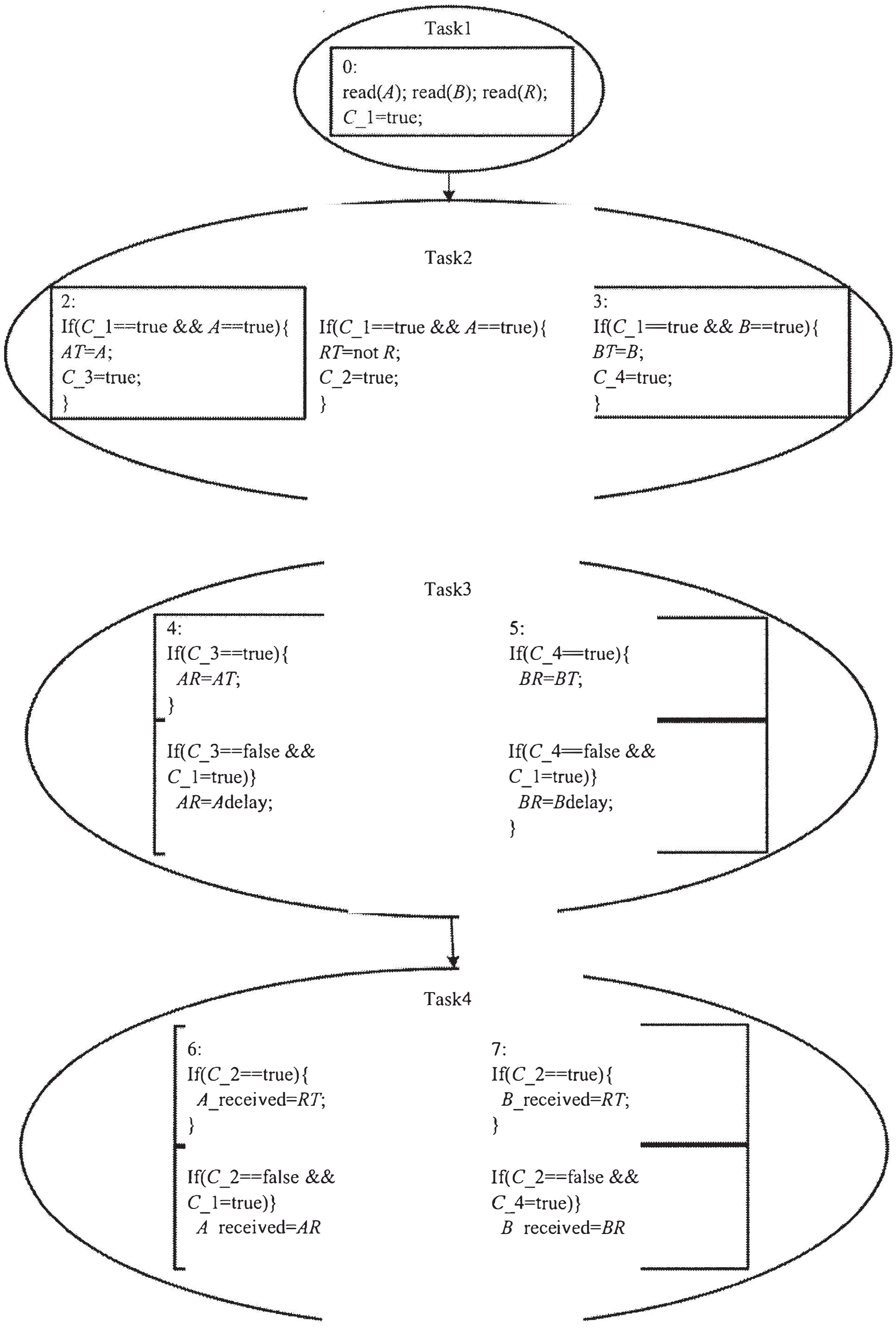


Fig. 5 Task partitions of DG in Fig. 4

```
#pragma omp directives [clause[clause]...]
```

Fig. 6 Syntax of OpenMP directive

and when all threads finish their executions, initial variables will be updated according to the calculation among its duplicates. However, every signal will be defined only once in the source program so that parallel nodes in the same task share no common variables. Consequently, the race condition will not appear in the generated code.

We take the form of the infinite loop of elementary iterations from [14] as the structure of the generated code. Tasks will be generated into the OpenMP structure as the core of the iteration. Here we only give the method translating tasks. Firstly, every node are translated into C code block. Secondly,

```
#pragma omp sections [clause[[,] clause] ...]

[#pragma omp section]
    structured-block
[#pragma omp section
    structured-block
```

Fig. 7 Syntax of directive parallel sections

each translated code block is encircled in the directive "#pragma omp section". Then, all blocks belonging to the same task will be encircled by directive "#pragma omp parallel sections". Finally, the sequential order of these blocks will be determined according to the total order specified among tasks. Note that the initial and terminal node of DG are respectively put in the front and the rear of the iteration and initial assignments to the memory signals should be put before the iteration part.

According to the method given above, fragment of the program in Fig. 3 can be translated to the OpenMP structure, shown in Fig. 8. We can see that sequential code is generated according to the order of the tasks. Since task2, task3, and task4 hare multiple code blocks, the corresponding OpenMP directives "#pragma omp parallel sections" are respectively generated. Code blocks which can be executed in parallel are encircled in the directive "#pragma omp section". Note that although assignments to the memory signal *A*delay and *B*delay are not shown in Fig. 8, the value of these two signals can be determined when they are on the right-hand side of the assignment statement, as Section 3.2 has indicated.

```
read(A); read(B); read(R);
C_1=true;
#pragma omp parallel sections{
    #pragma omp parallel section{
    if(C_1==true && A==true){
        AT=A;
        C_3=true;}}
    #pragma omp parallel section{
    if(C_1==true && B==true){
        BT=B;
        C_4=true;}}
    #pragma omp parallel section{
    if(C_1==true && R==true){
        RT=not R;
        C_2=true;}}

#pragma omp parallel sections{
    #pragma omp parallel section{
    if(C_3==true){
        AR=AT;}
    if(C_3==false && C_1==true){
        AR=Adelay;}}
    #pragma omp parallel section{
    if(C_4==true){
        BR=BT;}
    if(C_4==false && C_1==true){
        BR=Bdelay;}}

#pragma omp parallel sections{
    #pragma omp parallel section{
    if(C_2==true){
        A_received=RT;}
    if(C_2==false && C_1==true){
        A_recevied=AR;}}
    #pragma omp parallel section{
    if(C_2==true){
        B_received=RT;}
    if(C_2==false && C_4==true){
        B_received=BR;}}
```

Fig. 8 OpenMP-based C code corresponding to the Signal program fragment in Fig. 3

6 Conclusion and future works

This paper presented a methodology transforming endochronous Signal program (using core syntax) to OpenMP-based C code. First, the translation of Signal core primitives to code blocks of synchronous guarded action was described. Then, the formal definition of DG was presented, used to explore the implicit concurrency. From DG, the definition of task was given. Nodes of DG can be partitioned into tasks. Tasks will be executed in sequence while in each task, nodes can be executed in parallel. Finally, the method translating tasks into OpenMP-based C code was introduced. Using the approach, the generated program can run on multi-core processors, increasing the utilization of computation resources. Moreover, since the generation target OpenMP is a multi-platform standard, few modifications are needed for multi-platform execution.

However, several improvements can be accomplished from the current study. For instance, the method proposed in this paper does not allow the parallel execution among tasks, which may restrict the possibility of generating more efficient code. Another problem is that the methodology does not support the transformation of weakly endochronous Signal program, which limits the practicality of the study. In the future work, we will study how to check weak endochrony and generate deterministic code from weakly endochronous programs. Moreover, Signal provides arrays of processes to handle data arrays which is suitable for parallel execution. To generate better OpenMP code for these features is also one of our objectives.

Acknowledgements This work was supported by the National Natural Science Foundation of China (Grant Nos. 61073013 and 61003017) and the Aviation Science Foundation of China (2012ZC51025). Grateful acknowledgement is made to Mr. Mamoun FILALI-AMINE, Prof. Jean-Paul BODE-

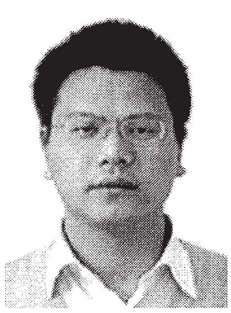
VEIX from IRIT-CNRS and Prof. Paul Le Guernic from INRIA. They have given a lot of instructive advice to this paper.

References

1. IEEE POSIX standardization authority. http://standards.ieee.org/regauth/posix/
2. Microsoft windows threads. http://msdn.microsoft.com/
3. MPI: A message-passing interface standard version 3.0. http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf
4. The OpenMP API specification for parallel programming. http://openmp.org/wp/
5. Intel thread building blocks. http://www.threadingbuildingblocks.org/
6. Lee E A. The problem with threads. Computer, 2006, 39(5): 33–42
7. Benveniste A, Berry G. The synchronous approach to reactive and real-time systems. Proceedings of the IEEE, 1991, 79(9): 1270–1282
8. Benveniste A, Caspi P, Edwards S A, Halbwachs N, Le Guernic P, De Simone R. The synchronous languages 12 years later. Proceedings of the IEEE, 2003, 91(1): 64–83
9. Berry G, Gonthier G. The esterel synchronous programming language: design, semantics, implementation. Science of Computer Programming, 1992, 19(2): 87–152
10. Halbwachs N, Caspi P, Raymond P, Pilaud D. The synchronous data flow programming language lustre. Proceedings of the IEEE, 1991, 79(9): 1305–1320
11. Le Guernic P, Gautier T, Le Borgne M, Le Maire C. Programming real-time applications with signal. Proceedings of the IEEE, 1991, 79(9): 1321–1336
12. Le Guernic P, Talpin J P, Le Lann J C. Polychrony for system design. Journal of Circuits, Systems, and Computers, 2003, 12(3): 261–303
13. Potop-Butucaru D, Caillaud B, Benveniste A. Concurrency in synchronous systems. Formal Methods in System Design, 2006, 28(2): 111–130
14. Besnard L, Gautier T, Talpin J P. Code generation strategies in the polychrony environment. http://hal.inria.fr/docs/00/37/24/12/PDF/RR-6894.pdf
15. Besnard L, Gautier T, Le Guernic P, Talpin J P. Compilation of polychronous data flow equations. In: Synthesis of Embedded Software, 1–40. Springer, 2010
16. Talpin J P, Ouy J, Gautier T, Besnard L, Le Guernic P. Compositional design of isochronous systems. Science of Computer Programming, 2012, 77(2): 113–128
17. 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
18. Jose B A, Shukla S K, Patel H D, Talpin J P. On the deterministic multi-threaded software synthesis from polychronous specifications. In: Proceedings of the 6th ACM/IEEE International Conference on Formal Methods and Models for Co-Design. 2008, 129–138
19. Maffeïs O, Le Guernic P. Combining dependability with architectural adaptability by means of the signal language. In: Static Analysis, 99–110. Springer, 1993
20. Potop-Butucaru D, Sorel Y, Simone d R, Talpin J P. From concurrent multi-clock programs to deterministic asynchronous implementations. Fundamenta Informaticae, 2011, 108(1): 91–118
21. Papailiopoulou V, Potop-Butucaru D, Sorel Y, Simone d R, Besnard L, Talpin J. From design-time concurrency to effective implementation parallelism: The multi-clock reactive case. In: Proceedings of the 2011 Electronic System Level Synthesis Conference. 2011, 1–6
22. Baudisch D, Brandt J, Schneider K. Multithreaded code from synchronous programs: extracting independent threads for openmp. In: Proceedings of the 2010 Conference on Design, Automation, and Test in Europe. 2010, 949–952
23. Besnard L, Gautier T, Le Guernic P. Signal v4-Inria version: reference manual, 2008
24. Gamatie A. Designing embedded systems with the signal programming language. Springer, 2010
25. RT-builder, geensys. http://www.geensys.com/
26. Polychrony. http://www.irisa.fr/espresso/Polychrony/

Kai Hu is an associate professor at Beihang University, China. He received his PhD 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.

Teng Zhang received his BE 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.



Zhibin Yang received his PhD degree from Beihang University, China, in February 2012. Since April 2012, he has been a Postdoc in IRIT research laboratory of University of Toulouse, France. His research interests include safety-critical real-time system, formal verification, AADL, and synchronous languages.