Verification of Synchronization in SpecC Description with the Use of Difference Decision Diagram

Thanyapat Sakunkonchak[†] and Masahiro Fujita

Department of Electronic Engineering, University of Tokyo
7-3-1, Hongo, Bunkyo, Tokyo, 113-8656, Japan
Phone:+(81)-3-5841-6764 Fax:+(81)-3-5841-6724
E-mail: thong@cad.t.u-tokyo.ac.jp[†], fujita@ee.t.u-tokyo.ac.jp

([†] denotes the contact author with the above address, phone and fax number.)

Abstract

SpecC language is designated to handle the design of entire system from specification to implementation and of hardware/software co-design. Concurrency is one of the features of SpecC which expresses the parallel execution of processes. Describing the systems which contain concurrent behaviors would have some data exchanging or transferring among them. Therefore, the synchronization semantics (`notify/wait`) of events should be incorporated. The actual design, which is usually sophisticated by its characteristic and functionalities, may contain a bunch of event synchronization codes. This will make the design difficult and time-consuming to verify. In this paper, we introduce an on-going work which helps verifying the synchronization of events in SpecC. The original SpecC code containing synchronization semantics is parsed and translated into a boolean SpecC code. The difference decision diagrams (DDDs) is used to verify for event synchronization on boolean SpecC code. The counter example for tracing back to the original source should be given when the verification result turns out to be unsatisfied. Here we introduce our overall idea and preset some preliminary results.

1. Introduction

Semiconductor technology has been growing rapidly, and entire systems can be realized on single LSIs as embedded systems or System-on-a-Chip (SoC). Designing SoC is a process of the whole system design flow from specification to implementation which is also a process of both hardware and software development. SpecC [1, 2] has been proposed as the standard system-level design language based on C programming language which covers the design levels from specification to behaviors. It can describe both software and hardware seamlessly and a useful tool for rapid prototyping as well.

Concurrency is becoming common-exist in describing a system design, both from the hardware and software aspects. The collaboration of parallel execution of behaviors/processes is fundamental to meet the design requirement and such collaboration is properly accomplished by realizing with the synchronization of those behaviors/processes. In a system design, synchronization might be oftenly exist and distributed throughout the design. Verifying the synchronization correctness in such a case may be difficult and significantly time-consuming.

This paper introduces an on-going work that tries to develop a technique for the verification of synchronization issues in SpecC language, a system level description language based on C. Recently the semantics of SpecC has been reviewed and clarified [11]. In this paper, we follow those semantics. In SpecC, expressing behaviors within semantic `par` results in parallel execution of those behaviors. For example, `par{a.main(); b.main();}` in Figure 1 implies that thread `a` and `b` are running concurrently (in parallel). Within behaviors, statements are running in the sequential manner just like C programming language. The timing constraint which must be satisfied for the behavior `a` is $Tas \leq T1s < T1e \leq T2s < T2e \leq Tae$, where notations s and e stand for starting and ending time, respectively. Note that it is not yet determined that any of "$st1 \rightarrow st2 \rightarrow st3$", "$st3 \rightarrow st1 \rightarrow st2$", and "$st1 \rightarrow st3 \rightarrow st2$" is being scheduled. In this

"All appropriate organizational approvals for the publication of this paper have been obtained. The author(s) will present the paper at the Forum"

case, an ambiguous result, or even worse, an access violation error could occur since $st1$ and $st3$ give the assignment value of the same variable x. The event manipulation statements, such as `notify/wait` could be applied in order to achieve the synchronization of any desired schedulings. `wait` statement suspends the current thread from execution until one of the specified events is `notify`.

```
main() {
   par{ a.main();
        b.main(); }}

behavior a {
   main() { x=10;        /*st1*/
            y=x+10;      /*st2*/ }}

behavior b {
   main() { x=20;        /*st3*/ }}
```

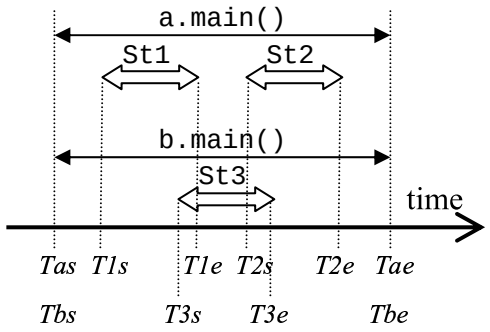

Figure 1: Timing diagram of the threads `a` and `b` under the `par{}`

The two parallel threads `a` and `b` as shown in Figure 2 where the synchronization statements of `notify/wait` is inserted into Figure 1. The statement `wait e` in thread `b` suspends the statement `st3` until the specified event `e` is notified. That is, it is guaranteed that statement `st3` is safely executed right after statement `st2`.

```
main() {
   par{ a.main();
        b.main(); }}

behavior a {
   main() { x=10;        /*st1*/
            y=x+10;      /*st2*/
            notify e;    /*New*/}}

behavior b {
   main() { wait e;      /*New*/
            x=20;        /*st3*/}}
```

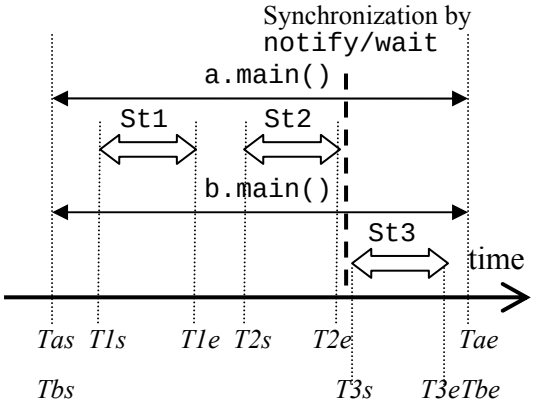

Figure 2: Insertion of synchronization statement `notify/wait` of Figure 1

With the timing diagram as shown in the previous figures, we can express the program statements in terms of inequalities of statements timing ($Tas \leq T1s < \ldots$). Hence, we can make use of the difference decision diagrams (DDDs) [9], a kind of the decision diagram [8] which can represent the inequalities efficiently, in order to verify the synchronization issues of the SpecC programs. SpecC programs are firstly parsed and translated into the boolean SpecC (the boolean programs which are generated from SpecC), then, translate those boolean SpecC into DDD graphs. Original idea of boolean programs were introduced by Ball and Rajamani [10]. The idea here is to abstract any conditions in `if` statements of the original programs with user-defined *predicates* and translate them into boolean domain. All statements other than event manipulation and conditions for `if` or `switch`, and so on, are removed (or abstracted away). Thus only boolean variables and event manipulation statements remain in the generated boolean programs. Here we use boolean programs as a kind of abstracted descriptions from original SpecC descriptions and verify them with DDDs, concentrating on verification of only synchronization issues in SpecC descriptions. Boolean variables are generated based on user-defined *predicates*, which define *abstraction functions* in verification process. Right now we are just assuming that *predicates* are given by designers (who are describing their designs in SpecC), but in the future we plan to develop automatic generation of *predicates* as well.

When verifying the synchronization of SpecC with DDDs, if the result turns out to be true, then

verification terminates and the synchronization is satisfied. When the result is false, however, the counter-example must be provided. This counter-example gives the trace back to the unsatisfied source in the original program.

2. Background

In this section, we give an overview of SpecC language, difference decision diagrams, and some basic concepts of the boolean programs. The concepts of sequentiality and concurrency are introduced. Semantics of `par` which describes the concurrency in SpecC is described as well as the event manipulator `notify/wait`.

2.1 SpecC Language

The SpecC language has been proposed as a standard system-level design language for adoption in industry and academia. It is promoted for standardization by the SpecC Technology Open Consortium (STOC, `http://www.SpecC.org`). The SpecC language was specifically developed to address the issues involved with system design, including both hardware and software. Built on top of C language, the de-facto standard for software development, SpecC supports additional concepts needed in hardware design and allows IP-centric modeling. Unlike other system-level languages, the SpecC language precisely covers the unique requirements for embedded systems design in an orthogonal manner.

Before clarifying the concurrency between statements, we have to define the semantics of sequentiality within a behavior. The definition is as follows. A behavior is defined on a time interval. Sequential statements in a behavior are also defined on time intervals which do not overlap one another and are within the behavior's interval. For example, in Figure 1, the beginning time and ending time of behavior `a` are Tas and Tae respectively, and those for `st1` and `st2` are $T1s$, $T1e$, $T2s$, and $T2e$. Then, the only constraint which must be satisfied is

$$Tas <= T1s < T1e <= T2s < T2e <= Tae$$

Statements in a behavior are executed sequentially but not always in continuous ways. That is, a gap may exist between Tas and $T1s$, $T1e$ and $T2s$, and $T2e$ and Tae. The lengths of these gaps are decided in non-deterministic way. Moreover, the lengths of intervals, ($T1e-T1s$) and ($T2e-T2s$) are non-deterministic as well.

Concurrency among behaviors are able to handle in SpecC with `par{}` and `notify/wait` semantics, see Figure 1 and 2. In a single-running of behaviors, correctness of the result is usually independent of the timing of its execution, and determined solely by the logical correctness of its functions. However, in the parallel-running behaviors, it is often the case that execution timing may have a great affect on the results' correctness. Results can be various depending on how the behaviors are interleaved. Therefore, the synchronization of events are important issue for the system-level design language. The definition of concurrency is as follows. The beginning and ending time of all the behaviors invoked by `par` statement are the same. Suppose the beginning and ending time of behavior `a` and `b` are Tas and Tae, and Tbs and Tbe, respectively. Then, the only constraint which must be satisfied is

$$Tas = Tbs,\ Tae = Tbe$$

According to these sequentiality and concurrency defined in SpecC language, all the constraints in Figure 1 description must be satisfied as follows.

- $Tas <= T1s < T1e <= T2s < T2e <= Tae$
  (sequentiality in `a`)
- $Tbs <= T3s < T3e <= Tbe$
  (sequentiality in `b`)
- $Tas = Tbs,\ Tae = Tbe$
  (concurrency between `a` and `b`)

The `notify/wait` statements are used for synchronization. `wait` statements suspends their current behavior from execution and keep waiting until one of the specified events is `notify`. Let focus on the `/*New*/` label in Figure 2 of which the event manipulation statements are inserted to that of Figure 1. We can see that `wait e` suspends `st3` until the event `e` is notified by `notify e`. As for the sequentiality, `notify e` is scheduled right after the completion of `st2` ($T2e <= T_{notify}s$). The only constraint for a single event synchronization is

$$T_{wait}e < T_{notify}s$$

2.2 Difference Decision Diagrams

As a part of formal verification, model checking [3] is extensively used to verify the system which can be expressed into finite states. McMillan [7] introduced the *symbolic* representation for the boolean variables which enhancing the use of decision diagrams, e.g. binary decision diagrams [8] (BDDs), to verify systems with very large number of states. However, if the constraints of model containing non-boolean, e.g. real-valued, variables, BDDs or other kind of symbolic representations of boolean variables are likely to be inefficient.

The idea of DDDs was introduced by Mϕller, *et al.* [9]. Its properties are mostly similar to that of BDDs except that it could handle the difference constraints, i.e. inequalities of the form $x - y \leq c$, where x and y are integer or real-valued variables and c is a constant. Figure 3 shows a DDD graph for $\neg(x - z < 1) \wedge (x - y \leq 0) \wedge (y - z \leq 2)$. DDDs share many properties with BDDs: 1) they are ordered, 2) they can be reduced making it possible to check for tautology and satisfiability in constant time, and 3) many of the algorithms and techniques for BDDs can be generalized to apply to DDDs. We use these inequalities to represent relating execution timings of event manipulation statements, and use boolean variables to represent control flows in the SpecC descriptions.

The size of the DDD graphs grow exponentially as the number of nodes increases. The current implementation of the DDD package claims that the it can handle the design up to 2048 variables. However, as we are trying to check for the capacity that DDD could handle, the memories have been used up before it reaches those limit of 2048 variables.

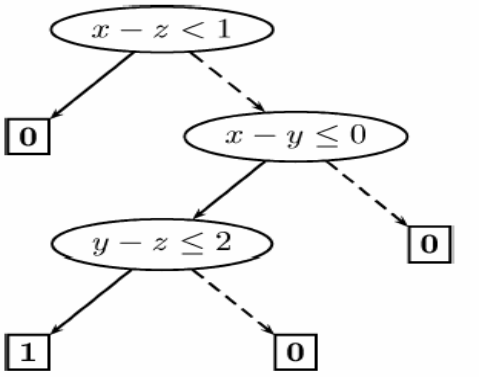

Figure 3: Difference decision diagram

2.3 Boolean Program

The work on boolean programs has been developing by conduction of Ball and Rajamani under the **S**oftware (Specifications), **L**anguages, **A**nalysis, and **M**odel checking (SLAM) project at Microsoft Research [10]. They try to conduct the verification on software by realizing the software as a model such that similar to the hardware FSM. This is to make the software model concrete to be verified by using the idea of model checking [3, 7]. The boolean programs have proved to be a subset of the original programs. What distinguishes boolean programs from FSMs is that the boolean programs contain procedures with recursion.

As the characteristic that boolean programs abstract the programs defined by the source language, the satisfied result of verifying some properties on the boolean programs ensures that those properties are satisfied the original programs as well.

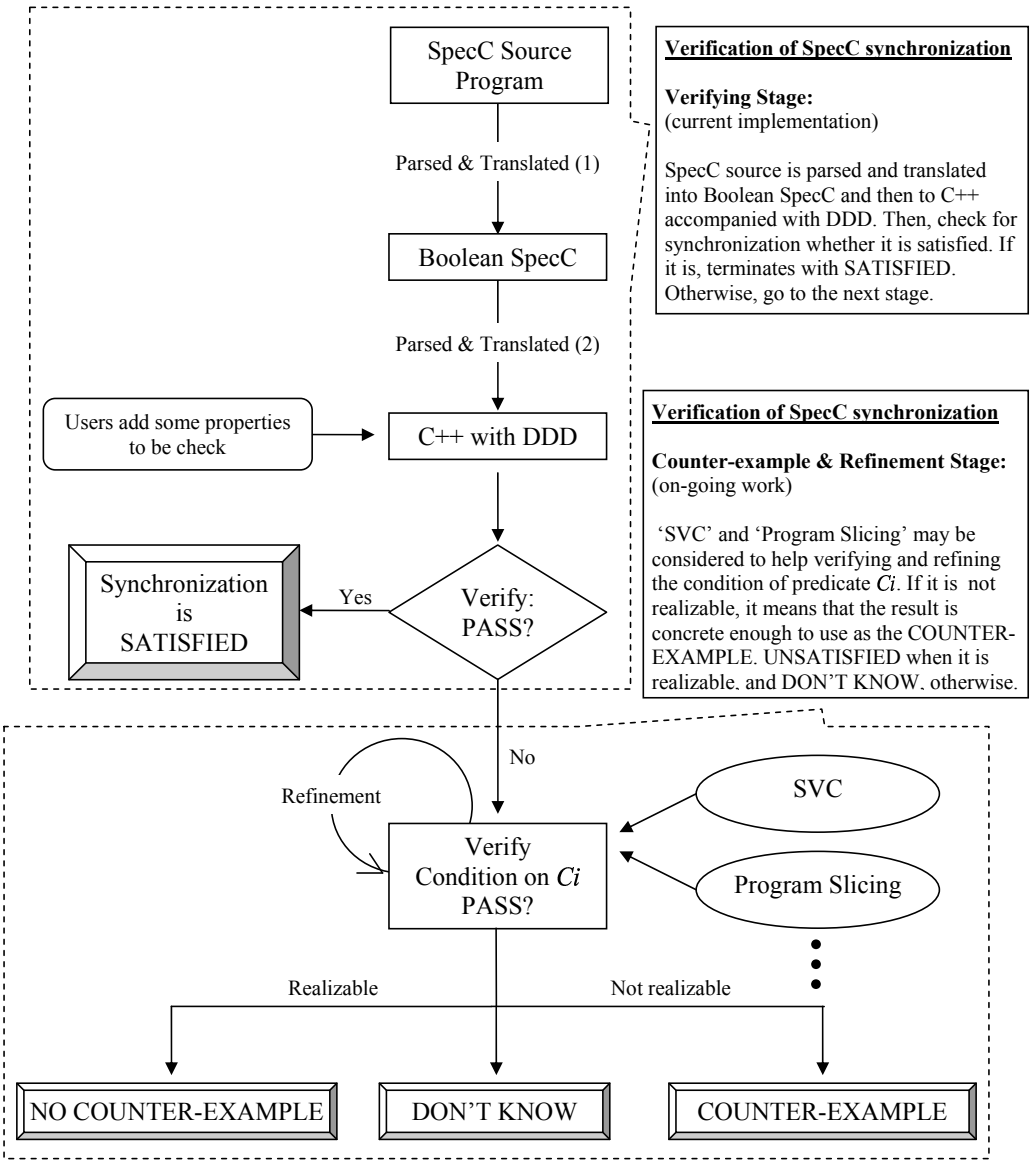

Figure 4: The proposed verification flow

3. Verification Flow

As we mentioned earlier that in both hardware and software system design, the concurrency is commonly appeared throughout the design description. The simple forms of synchronization which are realized in the design may not take the developers too many efforts to verify. Unfortunately, in many cases, the sophisticated forms of synchronization are likely to be appeared. Manually verifying the synchronization's correctness of the design would be difficult and exhausted. In this section, we propose the verification flow of synchronization in SpecC. The proposed verification flow is shown in Figure 4.

Our goal is to check whether the given SpecC codes containing concurrent statements `par` and event manipulation statements `notify/wait` are properly synchronized. We use the idea of the boolean programs, which represents a subset of the programs defined by the source language, in order to verify for the synchronization of events in SpecC. The flow can be roughly classified into two main stages: verifying and

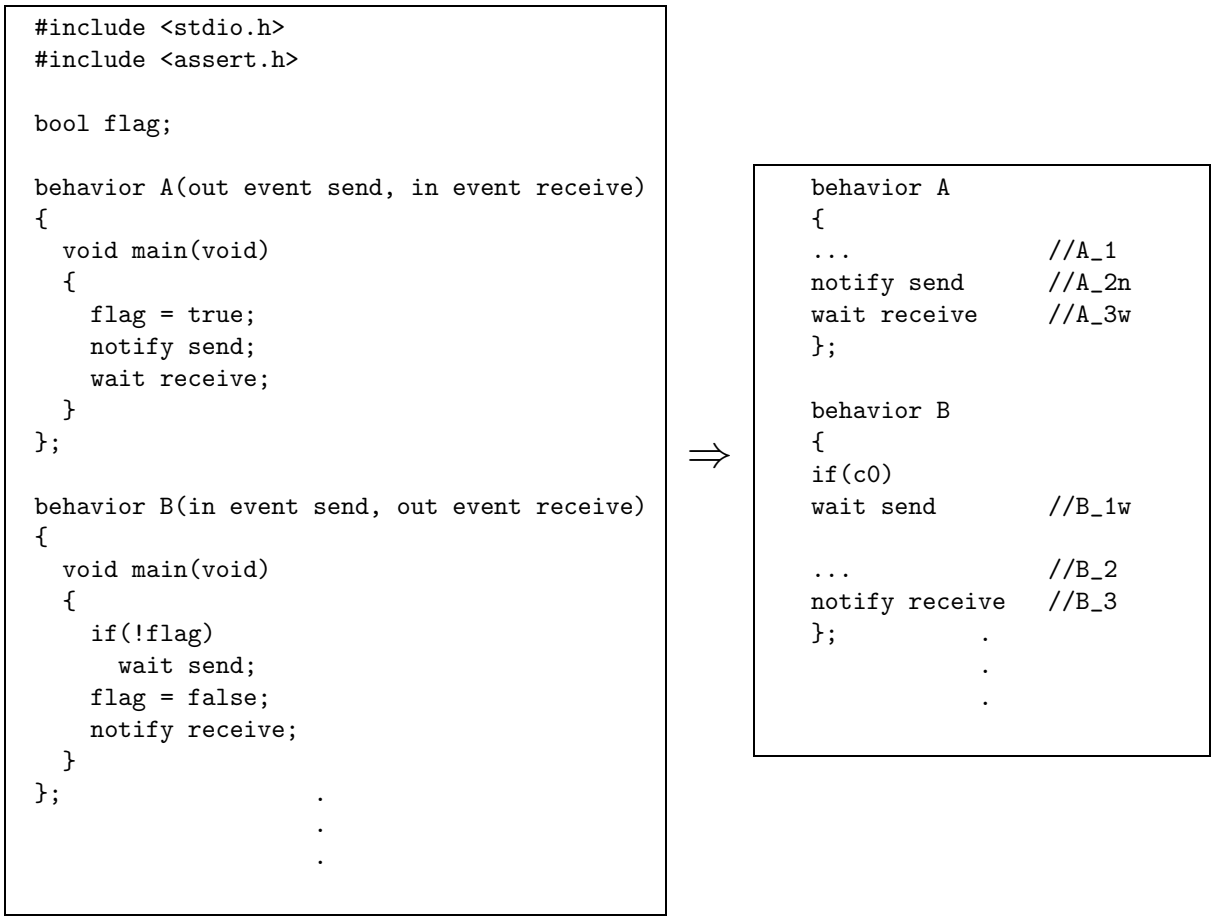

Figure 5: Translation from SpecC code to boolean SpecC

counter-example & refinement stage. Note that the current status of this work is at the verifying stage. We are planning to implement the rest as described in counter-example & refinement stage.

3.1 Verifying Stage (current implementation)

First, the SpecC source code must be parsed and translated into boolean SpecC code. The boolean SpecC code contains only conditional (`if` or `switch`) and event manipulation statements. Second, the achieving boolean SpecC is then parsed and translated into the C++ code which will be incorporate with the DDD package to verify for the event synchronization.

3.1.1 From SpecC to Boolean SpecC

The boolean programs [10] was proposed for software model checking. It is shown that the model itself is expressive enough to capture the core properties of programs and is amenable to model checking. A similar idea to the boolean programs is realized to verify for the SpecC synchronization. Let us assume that the original SpecC code to be verified is free of SpecC compilation and syntax errors (let the SpecC language compiler handle this). This is to avoid an undesirable results that will occur due to those errors. Then, the SpecC source code is parsing and translating such that

1) the event manipulation statements are sustained,
2) the conditional statements or predicates of all branching statements are automatically replaced by dummy variables, e.g. `if(x > 0)` is replaced by `if(c0)`, `if(x > 4)` by `if(c1)`, and so on,
3) all other statements are abstracted away by replacing with **skip** (denote in the boolean SpecC by "..." for readability).

Figure 5 gives an example of translation from SpecC original code to boolean SpecC code. At the current stage of implementation, we somehow cannot cover all possibilities of parsing and translating the SpecC source. For examples, statements contain any of the followings may not be able to parsed and translated.

- `for/do/do-while` loops
- complex `if-else` statements collaborate with loops, or vice versa
- recursive functions
- functions with passing variables

We are planning on working for the aboves. Let us note that there is a unique label for every statement. These labels will be captured as the variables when we translate from boolean SpecC to C++ file.

3.1.2 From Boolean SpecC to C++ with DDD

When achieving the correct parsed and translated boolean SpecC code, we again parse and translate to get the outcome in C++ code. Figure 6 gives the C++ code which translated from boolean SpecC in Figure 5. The structure of the generated C++ with enhancement of DDD package are described as followed (from the top down to the bottom of the generated code, respectively).

1) C++ header library where `"dddcpp.h"` denotes the DDD package library
2) `'ITE()'` function which handle the `if/else` branching [1]
3) Declaration of `'boolean'` and `'real'` variables. `A_1_a` and `A_1_b` are representing the beginning and ending time of the statement denoted by `A_1`
4) DDD graphs are generated which represent
   4.1) sequentiality in each behavior (`A` and `B`)
   4.2) concurrency among behaviors (`init_AB`)
   4.3) constraints of event (`send` and `receive`)
   4.4) bundle of every DDDs mentioned above (`init_AB_send_receive`)
5) Verify for the synchronization using DDD's `'Satisfiable'` function

3.1.3 Verifying with DDD

The achieved C++ outcome is then collaborated with DDD package and compiled with C++ compiler to verify for the event synchronization. DDD package provides an ability to check for the satisfiability of the DDD graphs which called `'Satisfiable'` function. Users can add properties to be checked. Consider the boolean SpecC in the right box of Figure 5. User may want to check whether the statement labeled `B_1w` (`wait send`) is reachable, for example.

The verification result should be 1) true, terminates all processes and returns the synchronization is satisfied or 2) otherwise, continues the process of finding the counter-example.

3.2 Counter-example & Refinement Stage (future plan)

All conditions of `if` statements in previous stage are abstracted into propositional boolean variables (Ci) and we verify for the synchronization without considering about the relationship among those abstracted predicates. Now, we are going to take all those predicates into account to further check whether the unsatisfied result from previous verifying stage can have a kind of counter-example to trace for errors in the source program. In verification at this stage, we are planning to use the following tools to verify and make a refinement of those predicates.

[1] To represent the inequalities as DDD nodes (see Figure 3), the conventional C/C++ data types cannot be used to represent these nodes. The new data types `'boolean'` and `'real'` are introduced. `'ITE()'` function is, therefore, similarly performed like conventional `if/else` statements.

```
/*****************************************************************/
/* 1) Library header "dddcpp.h" is the DDD library package for C++ */
/*****************************************************************/
#include<dddcpp.h>
#include<iostream.h>
#include<stdio.h>
#include<time.h>

/************************************************/
/* 2) if/else branching function for DDD expression */
/************************************************/
ddd ITE(const ddd& test_expr, const ddd& iftrue, const ddd& iffalse){
  return(test_expr & iftrue) } (test_expr & iffalse);
}

int main(){
  /******************************************************/
  /* 3) Declare variables with 'boolean' and 'real' type */
  /******************************************************/
  ddd::Init();
  boolean c0="c0";
  real A_1_a="A_1_a",A_1_b="A_1_b",A_2n_a="A_2n_a",A_2n_b="A_2n_b",A_3w_a="A_3w_a",A_3w_b="A_3w_b";
  real B_1w_a="B_1w_a",B_1w_b="B_1w_b",B_2_a="B_2_a",B_2_b="B_2_b",B_3n_a="B_3n_a",B_3n_b="B_3n_b";

  /*************************************************************************************/
  /* 4) Generate DDD graphs corresponding to sequentiality and concurrency among behaviors */
  /*************************************************************************************/
  ddd A = (A_1_a-A_1_b<0)&(A_1_b-A_2n_a<=0)&(A_2n_a-A_2n_b<0)&(A_2n_b-A_3w_a<=0)&(A_3w_a-A_3w_b<0);
  ddd B = ITE(c0,(B_1w_a-B_1w_b<0),False)&(B_1w_b-B_2_a<=0)&(B_2_a-B_2_b<0)&(B_2_b-B_3n_a<=0)&
          (B_3n_a-B_3n_b<0);
  ddd init_AB = (!(A_1_a-B_1w_a<0))&(A_1_a-B_1w_a<=0)&(!(A_3w_b-B_3n_b<0))&(A_3w_b-B_3n_b<=0);
  ddd send = (B_1w_a-A_2n_b<0);
  ddd receive = (A_3w_a-B_3n_b<0);
  ddd init_AB_send_receive = init_AB & A & B & send & receive;

  /*************************************************************************/
  /* 5) Verify for synchronization of all events with 'SATISFIABLE' function */
  /*************************************************************************/
  if(Satisfiable(init_AB_send_receive))
    cout << "Synchronization is SATISFIED\n";
  else
    cout << "Synchronization is UNSATISFIED\n";

  return(0);
}
```

Figure 6: Translation from boolean SpecC code to C++ with enhanced DDDs

- **Standford Validity Checker (SVC):** It is a tool used for validity checking of the boolean formulas. It is proved to be efficient. We will use it for checking for the correctness of the decision procedure.
- **Program Slicing:** In general, it is served for finding statements that potentially affect the computation of a specific variable at specific statement. Here we will utilize for some variables tracing.

The conditions on Ci can be considered as:

1) "Realizable". This will result in the unsatisfied of the synchronization. The result that we got here cannot be used as the counter-example for error tracing.
2) "Not realizable". This means that the result that we have got can really be a counter-example

for tracing to the source that give the unsatisfied result.

3) "Don't know". The process terminates with "don't know" answer.

4. Experimental Results

The current status of the implementation of this work, as shown in the verification stage in Figure 4, still cannot fully verify the real application programs which usually contain every type of language semantics and complex data structures. We then try to come up with a simple synchronization example which given in many OS textbooks [4, 5, 6].

> **Sleeping (daydreaming) barber problem**
> *The shop has a barber, a barber chair, and a waiting room with N chairs. When a barber finishes cutting a customer's hair, the barber fetches another customer from the waiting room if there is a customer, or stands by the barber chair and daydreams if the waiting room is empty. A customer who needs a haircut enters the waiting room. If the barber is busy but there is a waiting room chair available, the customer takes a seat. If the waiting room is empty and the barber is daydreaming, the customer sits in the barber chair and wakes up the barber.*

We start verifying for the SpecC description of this example. Part of SpecC description and boolean SpecC that describe the flows of the barber and the customers are shown in Figure 7. The entire processes from parsing and translating the SpecC source to verifying for the synchronization of events are automatic. The results of verifying some properties in the above example are shown in Table 1. We use the `'Satisfiable'` function of the DDD package to verify and check with a specific property.

The result of verifying the SpecC description as shown in Table 1 can be explained in details as followed:

"SATISFIED"	With the property `Customer_7 > Barber_4`, it is obviously seen that the result of verification of this property must be satisfied. Our program terminates with the satisfied result.
"UNSATISFIED"	This property, similar to the above, must give the verification result as satisfied if the SpecC source was properly synchronized. Let consider the code in Figure 7, on the line number 19-20 and 43-44. Even the synchronization of the individual event is correct, with these coding style, it means 'the customer will leave before the cutting finished'. The exchanged position between line number 19⇆20 and 43⇆44 should satisfy the checking property.
"DON'T KNOW"	Again, the SpecC synchronization description should be satisfied by this property `Barber_1 > Customer_5`. However, since the statement `Barber_1` and `Customer_5` are under the `if` statements of dummy variables `c0` and `c3`, the verification result has not been knowing yet unless the relationship between those variables is clarified.

With the case of "UNSATISFIED" and "DON'T KNOW" results, the current implementation is still cannot neither trace back to the unsatisfied source nor verify for the relationship among the `if` conditions. To make this work able to completely verify for the synchronization, we are planning to work on making the counter-example and verifying the dummy variables which abstracted predicates, as described in previous section.

5. Conclusion and Outlook

We proposed the technique for verifying the synchronization of events in SpecC descriptions with the use of DDDs which is amenable to express the different constraints. The concept of the boolean programs is applied to abstract away some details other than the event manipulation and conditional branching

Table 1: Example of checking some properties on the sleeping barber problem.

Properties to be checked	**Verification results**
The barber chair is empty after the cutting is finished (`Customer_7 > Barber_4`)	SATISFIED
The customer leave the chair when the barber finished cutting (`Customer_5 > Barber_4`)	UNSATISFIED
The barber waits for a new customer when the previous one leave (`Barber_1 > Customer_5`)	DON'T KNOW

statements. The SpecC code can be checked for the correctness of the event synchronization and let users be able to give some constraints to invoke with the original model from SpecC code. However, up to this point, there are still some limitations on handling the original SpecC code, e.g. the looping, passing variables to a function, recursive functions may not be properly parsed and translated for verification. We are also planning to provide the counter-example and the way to verify it by collaborating with some tools like SVC and Program Slicing.

As a final remark, the proposed technique clearly defines synchronization semantics of SpecC descriptions, which is one of most important issues for system level description languages.

References

[1] D. Gajski, J. Zhu, R. Doemer, A. Gerstlauer, and S. Zhao, SpecC: Specification Language and Methodology, *Kluwer Academic Publisher*, March 2000.

[2] A. Gerstlauer, R. Doemer, J. Peng, and D. Gajski, System Design: A Practical Guide with SpecC, *Kluwer Academic Publisher*, June 2001.

[3] E. M. Clarke, O. Grumberg, and D. Peled, Model Checking, *MIT Press*, January 2000.

[4] G. R. Andrews, Concurrent Programming: Principles and Practice, *Addison-Wesley*, 1991.

[5] G. R. Andrews, Foundations of Multithreaded, Parallel, and Distributed Programming, *Addison-Wesley*, 2000.

[6] S. Hartley, Concurrent Programming - The Java Programming Language, *Oxford University Press*, 1998.

[7] K. L. McMillan, Symbolic Model Checking, *Kluwer Academic Publishing*, July 1993.

[8] R. E. Bryant, "Graph-based Algorithms for Boolean Function Manipulation," *IEEE Transactions on Computers.*, Vol. C-35, No. 8, pp. 677-691, August 1986.

[9] J. Mϕller, J. Lichtenberg, H. R. Anderson, and H. Hulgaard, "Difference Decision Diagrams," *Technical report IT-TR-1999-023, Department of Information Technology, Technical University of Denmark.*, February 1999.

[10] T. Ball and S. K. Rajamani, "Boolean Programs: A Model and Process For Software Analysis," Microsoft Research, http://research.microsoft.com/slam

[11] M. Fujita and H. Nakamura, "The Standard SpecC language," *Proc. of ISSS 2001*, Montreal, Canada, October 2001.

```
Line#
  1   #include <stdio.h>
  2   #include <assert.h>
  3
  4   const int N = 5;
  5   bool CustomerMeetBarber = false;
  6   bool BarberChairEmpty = true;
  7
  8   behavior Barber(in event CustomerInBarber,
  9                   out event Finish,
  10                  in event Leave)
  11  {
  12    void main(void)
  13    {
  14      if(BarberChairEmpty == true)
  15        wait    CustomerInBarber;
  16
  17      if(CustomerMeetBarber == true){
  18        cutting();
  19        wait    Leave;     // Should be line# 20
  20        notify  Finish;    // Should be line# 19
  21        CustomerMeetBarber = false;
  22      }
  23    }
  24  };
  25
  26  behavior Customer(out event CustomerInBarber,
  27                    in event Finish,
  28                    out event Leave,
  29                    inout int numCustomers)
  30  {
  31    void main(void)
  32    {
  33      if(numCustomers == N)
  34        CustomerGoback();
  35      else
  36        numCustomers++;
  37
  38      if(BarberRoomEmpty == true){
  39        notify  CustomerInBarber;
  40        numCustomers--;
  41        BarberChairEmpty = false;
  42        CustomerMeetBarber = true;
  43        notify  Leave;     // Should be line# 44
  44        wait    Finish;    // Should be line# 43
  45        BarberChairEmpty = true;
  46      }
  47    }
  48  };                .
                        .
                        .
```

⇒

```
behavior Barber
{
if(c0){
wait CustomerInBarber   //Barber_1

}
if(c1){
cutting()               //Barber_2
wait    Leave           //Barber_3
notify  Finish          //Barber_4
...                     //Barber_5

}
};

behavior Customer
{
if(c3){
notify CustomerInBarber //Customer_1
...                     //Customer_2
...                     //Customer_3
...                     //Customer_4
notify  Leave           //Customer_5
wait    Finish          //Customer_6
...                     //Customer_7

}
};              .
                .
                .
```

Figure 7: SpecC description & boolean SpecC of barber and customer thread on the sleeping barber problem