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

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. In this paper, we introduce an on-going work of 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. 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.

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. 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 $Tas \leq T1s < T1e \leq T2s < T2e \leq Tae$. 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 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`.

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`.

Synchronization is nothing but the scheduling of the statements which can be expressed in terms of the inequalities of execution timings of statements ($Tas \leq T1s < \ldots$). Hence, we make the use of the difference decision diagrams (DDDs), a kind of the decision diagram 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 program which is generated from SpecC), then, translate those boolean SpecC into DDD graphs. Original idea of boolean program was introduced by Ball and Rajamani [7]. The idea here is to abstract any conditionals in `if` statements of the original programs with user-defined *predicates* and translate them into boolean domain. All statements other than event manipulation and conditionals 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 program as a kind of abstracted descriptions from original SpecC descriptions and verify them with DDDs, concentrating on verification of only synchronization issues in SpecC

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

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

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

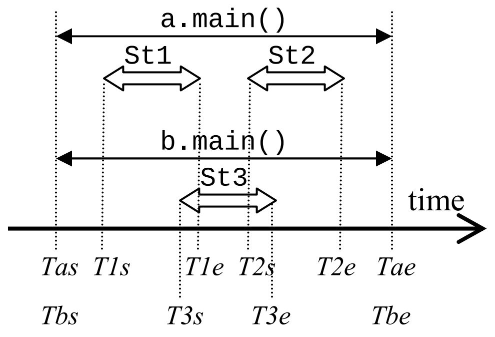

Figure 1. Timing diagram of the threads `a` and `b` under the `par{}`

```
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*/}}
```

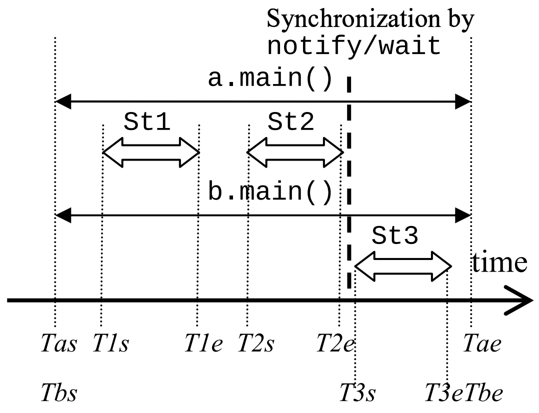

Figure 2. Insertion of synchronization statement `notify/wait` of Figure 1

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.

2. Background

In this section, we give an overview of SpecC language and difference decision diagrams. 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). 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

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

Statement 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 [4] introduced the *symbolic* representation for the boolean variables which enhancing the use of decision diagrams, e.g. binary decision diagrams [5] (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.

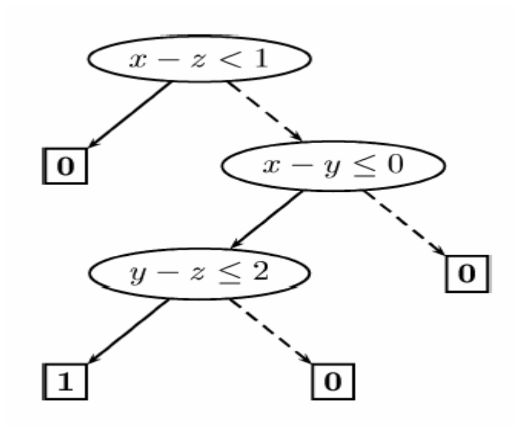

Figure 3. Difference decision diagram

The idea of DDDs was introduced by Mϕller, *et al.* [6]. 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.

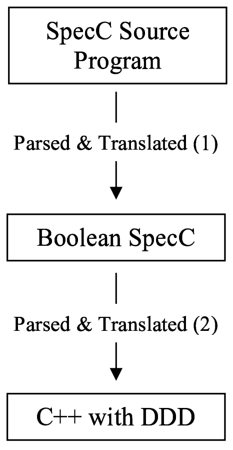

Figure 4. Verification flow

3. Verification Flow

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 program, which represents a subset of the program defined by the source language, in

```
#include <stdio.h>
#include <assert.h>

bool flag;

behavior A(out event send, in event receive)
{
  void main(void)
  {
    flag = true;
    notify send;
    wait receive;
  }
};

behavior B(in event send, out event receive)
{
  void main(void)
  {
    if(!flag)
      wait send;
    flag = false;
    notify receive;
  }
};                    .
                      .
                      .
```

$\Rightarrow$

```
behavior A
{
...              //A_1
notify send      //A_2n
wait receive     //A_3w
};

behavior B
{
if(c0)
wait send        //B_1w

...              //B_2
notify receive   //B_3
};          .
            .
            .
```

Figure 5. Translation from SpecC code to boolean SpecC

order to verify for the synchronization of events in SpecC. 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. The verification flow is shown in Figure 4.

3.1 From SpecC to Boolean SpecC

The boolean program [7] 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 program 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 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.

3.2 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`)

[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

4.4) bundle of every DDDs mentioned above (`init_AB_send_receive`)

5) Verify for the synchronization using DDD's ‘`Satisfiable`’ function

The achieved C++ outcome is then compiled with C++ compiler, e.g. g++, to verify for the event synchronization. DDD package provides an ability to check for the satisfiability of the DDD graphs which called ‘`Satisfiable`’ function.

4. Verification Results

The entire verification flow that we have described, from capturing the original SpecC code and translating to boolean SpecC, to translating the obtained boolean SpecC into the C++ code with DDD enhancement, are automatically processed.

The result of the verification could be any of these, the synchronization of the original code is satisfied, not satisfied, or don't know. Note that for the case of ‘not satisfied’, it should be able to give a counter-example which can be used to track the unsatisfied source. Up to this stage, we can verify only the synchronization of events from SpecC code. An example of the verification flows from the original SpecC code to boolean SpecC as shown in Figure 5, from boolean SpecC to C++ code as shown in Figure 6, then, compile and execute with the C++ compiler. The result shows that the original code is satisfied if the event manipulation statements are well defined. Let see an example of Figure 7. Among thread `a` and `b`, the conditions for an event `e` to be synchronized are that `(x < y)` and `(x >= y)` must be true at the same time. In this case, the original code has no chance to be synchronized.

We are planning to add the abilities

1) for users to interactively input other constraints to check with the original source
2) to the predicates that we abstracted from `if`

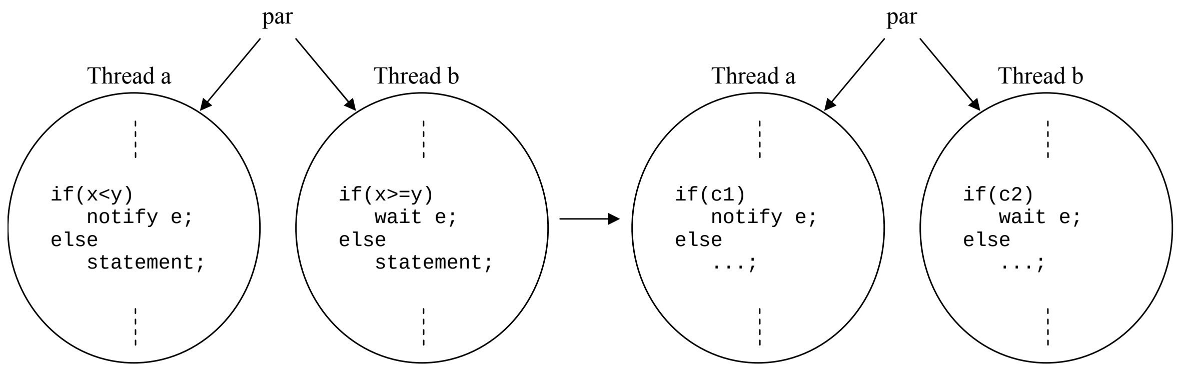

Figure 7. An example of parsing SpecC code to boolean SpecC

or `switch` statements and provide the counter-examples when verification is not satisfied

3) to be able to automatically generate the 'predicates' to avoid intentional errors that will cause from users
4) of refinement of predicates or to add constraints on predicates corresponding to the original SpecC code

Figure 8 shows the complete verification flow that we expect to achieve.

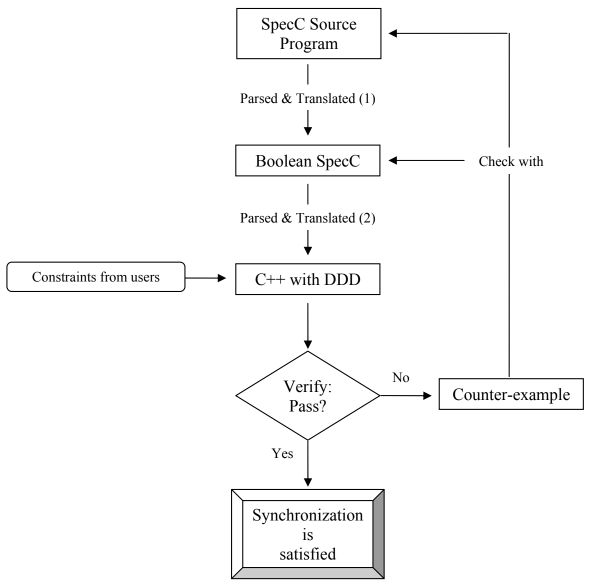

Figure 8. Flow of verification of synchronization with abilities to accept constraints from users and provide counter-example

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 program is applied to abstract away some details other than the event manipulation and conditional branching statements. Up to present, the SpecC code can be checked for the correctness of the event synchronization. We are planning to let users be able to give some constraints to invoke with the original model from SpecC code. In addition, the conditional for `if` or `switch` (or *predicates*) should be automatically generated rather than defined by users.

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] K. L. McMillan, Symbolic Model Checking, *Kluwer Academic Publishing*, July 1993.
[5] R. E. Bryant, "Graph-based Algorithms for Boolean Function Manipulation," *IEEE Transactions on Computers.*, Vol. C-35, No. 8, pp. 677-691, August 1986.
[6] 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.
[7] T. Ball and S. K. Rajamani, "Boolean Programs: A Model and Process For Software Analysis," Microsoft Research, http://research.microsoft.com/slam