Formal Verification of Synchronization Issues in SpecC Description with Automatic Abstraction

Thanyapat Sakunkonchak and Masahiro Fujita
Department of Electronic Engineering, University of Tokyo
7-3-1 Hongo, Bunkyo-ku, Tokyo 113-8656 JAPAN
Phone&Fax: +(81)-3-5841-6764
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. 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 a technique 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 examples for tracing back to the original source are given when the verification results turn out to be unsatisfied. Here we also introduce idea on automatically refinement when the results are unsatisfied 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. 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.

The size of the design is one of the major problems in the verification field. When the design becomes larger, it is usually unable to verify due to the memory explosion or the limitation of the capability of the verification tools. Boolean programs [10] are the software verification technique that use the idea of the abstraction of the original program codes, namely those in which all variables and parameters have boolean type. Verification of the boolean programs, with the fact that the boolean programs are the subsets of the original ones, if the results are satisfied, we can directly imply that the original program codes are also satisfied.

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. 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 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 scheduling. `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`.

In this paper, we develop and demonstrate a technique for the verification of synchronization issues in SpecC language. The proposed technique applies the idea of the

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

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

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

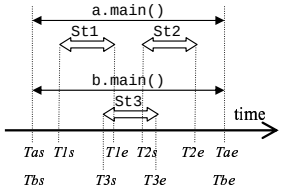

Fig. 1. Timing diagram of the threads `a` and `b` under the `par{}`

```
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*/}}
```

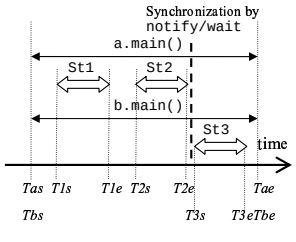

Fig. 2. Insertion of synchronization statement `notify/wait` of Figure 1

boolean program [10]. The abstracted SpecC code (to avoid confusion with the boolean programs, let us called 'boolean SpecC') is containing the statements that can be expressed in terms of inequalities of timing of those statements ($Tas \leq T1s < \ldots$). 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. 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. The idea for the automatically refinement of the predicates is proposed. We believe that the implementation of this refinement of predicates would help the designers to save a lot of time to find errors.

Next section will give some background on the SpecC synchronization semantics, DDDs and boolean programs. The proposed verification flow is introduced in section 3. Some conclusions are in section 4 where we mention also the future directions for this work as well.

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 (overflow) before it reaches those limit of 2048 variables.

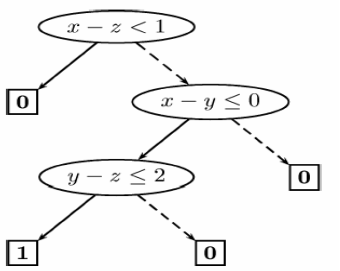

Fig. 3. Difference decision diagram

2.3: Boolean Program

The study of model checking has been an active area of research during the past few decades. The method of model checking was firstly applied for verifying the LSI circuit systems. The extensive study lead to significant new techniques, e.g. temporal logic and symbolic representations, which enhance the larger and more complex systems able to be verified. As the great achievement in applying the model checking for hardware, such technique has also been applying for software or even both hardware/software co-design.

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.

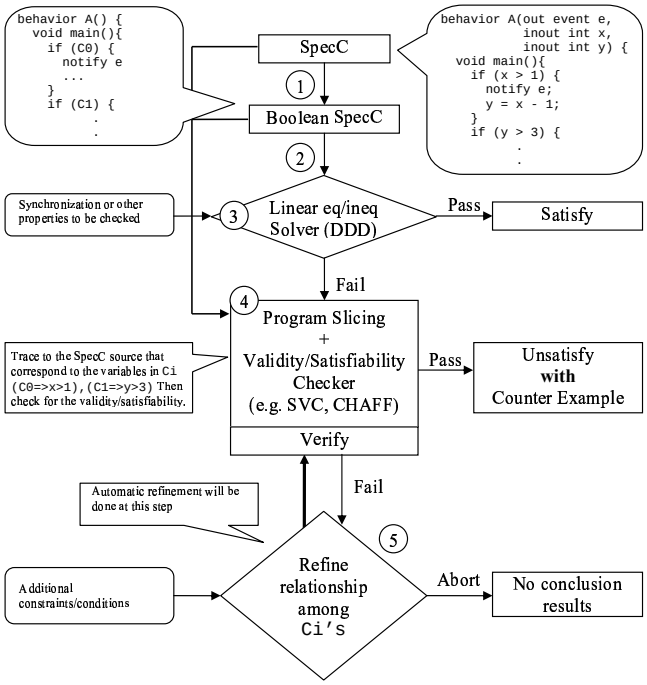

Fig. 4. The proposed verification flow

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 refinement stage. Note that the sub-section 3.1.1-3.1.4 for verifying stage are corresponding to the step ①-④ in Figure 4. Step ⑤ is to the refinement stage.

The implementation of the proposed technique is attempted to originally established to verify some simple designs. We are planning to cover the more complex designs, e.g. the designs that having recursive functions or nested loops, and to make to whole processes of verification for synchronization in SpecC, from verifying stage to refinement stage, automatically processed.

3.1: Verifying Stage

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.

1) From SpecC to Boolean SpecC: The boolean programs 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. This is to avoid an undesirable results that will occur due to those errors (let the SpecC language compiler handle this). 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 > 1)` is replaced by `if(C0)`, `if(y > 3)` 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)

Let us note again that the output at this step contains only the boolean variables. This may seem to be too much abstraction to skip every statement other than conditional and event manipulation statements. We will, however, show in the latter sections the way to de-abstract to get more information over those abstracted part by tracing over the original SpecC.

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. The DDD package version 2.0 is used to verify the synchronization of events in boolean SpecC. The structure of the generated C++ with enhancement of DDD package are constructed.

3) Verifying with DDD: Now we have the achieved C++ codes that can be verified using DDD package. The outcome is then 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.

The verification at this step is only to make a first check for any suspicious event that has a chance not to be synchronized, e.g. the event is seem to be 'wait' forever. Some properties to be verified can be accordingly added. The verification result should be either 1) true, terminates all processes and returns the synchronization is satisfied or

2) otherwise, continues the process of finding the counter-example.

In the process of finding such a counter-example, the validity/satisfiability checker is employed to check for the predicates that we abstracted with the boolean variables, e.g. `if(x > 1)` to `if(C0)`. The program slicing tool may also be used to trace on the original SpecC codes to find the causes of unsatisfiability.

4) Verifying with Validity/Satisfiability Checker: 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 using the following tools to verify and make a refinement of those predicates.

- **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.
- **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 is applied to slice to the parts of the codes that contain the statements that related to the variables in predicates, for example, the program slicing will use to slice to the statements that contain the assignment of variable `x` for predicate `x > 0`, and so on. All predicates which abstracted from all Ci and all related variables are validated using the SVC. The results for validating can be either of 1) true, and the program terminates with some counter-examples that witness the errors occur in the synchronization of events in the designs or 2) otherwise, the program can stop here and gives the result that the designs are unable to verify.

3.2: Refinement Stage

At this step, referring to step ⑤ in Figure 4, we propose the method for automatically refining the predicates. Remark that some constraints can be added to control some variables on predicates during the refining process. After modified, the validity checker is used to check the results again. If the results are satisfied, then program terminates with some counter-examples. However, if the results turn to be unsatisfied (false), the refining process keeps running until it reaches a threshold. Then, program aborts with no conclusion results. Of which, we do not really know whether the synchronization in the designs is correct or incorrect. In this case, the designers have to manually review and modify the designs by themselves.

We are planning to make the refinement process automatically operate. We do believe that the automatic refinement will save the designers a lot of time and efforts on finding some synchronization errors from the whole design.

4: 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 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.

The unsatisfied results of verifying boolean SpecC with DDDs will be further verified for the correctness of those abstracted predicates using program slicing and validity checker. Unfortunately, the tool that can slice through the SpecC code is not currently available. As the future work, we intend to make the process of refinement of predicates automatically run.

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.

[12] S. Honda and H. Takada, "Evaluation of The Description Capability of SpecC Through a Serial Device," *DA Symposium 2001*, June 2001.