社団法人　電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
TECHNICAL REPORT OF IEICE.

システムレベル設計における同期検証フレームワークの提案

サクンコンチャック タンヤパット[†]　小松　聡[†]　藤田　昌宏[†]

† 東京大学、大規模集積システム設計教育研究センター
E-mail: †{thong,komatsu}@cad.t.u-tokyo.ac.jp, ††fujita@ee.t.u-tokyo.ac.jp

A Framework on Synchronization Verification in System-Level Design

Thanyapat SAKUNKONCHAK[†], Satoshi KOMATSU[†], and Masahiro FUJITA[†]

† Takeda-Sentanchi Building, 2–11–16, Yayoi, Bunkyo-ku, Tokyo, 113–0032 Japan
E-mail: †{thong,komatsu}@cad.t.u-tokyo.ac.jp, ††fujita@ee.t.u-tokyo.ac.jp

Abstract This paper presents a formal verification framework targeting the synchronization problem in SpecC language, a C-based system-level design language that supports HW/SW design seamlessly. Based on the Counter-Example Guided Abstraction Refinement (CEGAR) paradigm, we can briefly describe our framework as follows. The SpecC descriptions are abstracted and translated into descriptions that contain only boolean variables, or so called "boolean SpecC". The boolean SpecC descriptions are translated into an equality/inequality formula. This formula is then checked using an Integer Linear Programming (ILP) solver. Once the results are satisfied, the verification process stops with the synchronization property holds. Otherwise, a counterexample is given. The process of counterexample analysis and abstraction refinement is also proposed in this paper.

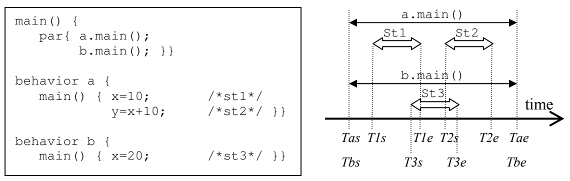
Key words formal verification, model checking, predicate abstraction, abstraction refinement, synchronization verification.

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. A language that supports hardware-software co-design and the ability to appropriately partition between hardware execution part and software execution part is needed. Recently, there are lots of attentions into the use of C programming language, since it is commonly used in the software development, C-based (or the extensions of C-based like SystemC or SpecC) SoC design is a promising approach to cover both hardware and software design with a single design/specification language.

SpecC language [8], [9] has been proposed as a standard system-level design language based on C programming language and adds new constructs and semantics for describing parallel behaviors, pipelined behaviors, finite state machine, and arbitrary-length bit-vectors. SpecC also gives the clear separation between computation and communication bodies in system-level descriptions. The design can be modularized by 1) the computation part can be handled by the construct called *behavior* and 2) the communication among processes are done through the *channel* construct. Our discussion is concerned the concurrency (*par* semantics) and synchronization (*notify/wait* semantics) in SpecC because the pipelined behaviors (*pipe* semantics) can be defined by using *par*.

In SpecC, expressing behaviors within *par* statement results in parallel execution of those behaviors. For example, *par*{*a.main(); b.main();*} in Fig. 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 constraints which must be satisfied for the behavior a is $Tas \leqq T1s < T1e \leqq T2s < T2e \leqq Tae$, where $T1$ and $T2$ stand for the timing of statements *st1* and *st2*, and the postfix notations s and e stand for starting and ending time, respectively. This shows the sequential execution of $st1$ and $st2$. Note that it is not yet de-

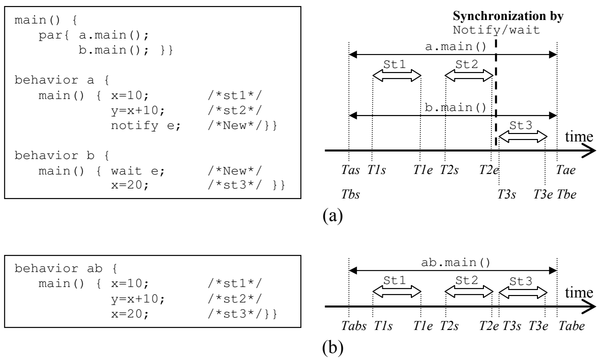


```
main() {
   par{ a.main();
        b.main(); }}

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

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

図 1 Timing diagram of the threads a and b under the $par\{\}$

termined 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 to the same variable x. The event manipulation statements in SpecC, *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 Fig. 2(a) where the synchronization statements of *notify/wait* is inserted into Fig. 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*. This eliminates the ambiguous results.



```
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*/ }}
```

```
behavior ab {
   main() { x=10;        /*st1*/
            y=x+10;      /*st2*/
            x=20;        /*st3*/}}
```

図 2 (a) Insertion of synchronization statement *notify/wait* of Fig. 1, (b) sequential description which is equivalent to the description in (a)

According to the successfulness of applying formal methods in the hardware design industry, recently there are many researches applying formal methods to verify the correctness of software, e.g. ANSI-C or Java language. *Model checking* [4], [14] is one of such techniques commonly used. However, the effectiveness of model checking for software is severely degraded due to the state explosion problem. Abstraction technique, particularly *predicate abstraction* [1], [10], [12], is used to reduce the state space by constructing an abstract model which is a subset of programs defined by the source language. The refinement of abstraction can be automated using the Counter-Example Guided Abstraction Refinement (CEGAR) paradigm [3].

Clarke et al. [5] show the verification of SpecC using predicate abstraction. SpecC descriptions is translated into FSMs with extra treatment on *notify/wait* for synchronization. When applying predicate abstraction to the concurrent programs, an interleaving of global or shared variables among those concurrent processes is needed to be treated as well. The size of the problem grows exponentially as the number of concurrent processes increases. When the number of concurrent processes is large, this can severely caused the state explosion even the abstraction technique is applied.

In this paper, a framework of synchronization verification in SpecC descriptions is introduced. Abstraction method is applied to reduce the size of the target systems. Once the synchronization correctness is verified, the further verification for the entire systems can be proceeded. For example with appropriate synchronization verification, one may be able to verify other properties, e.g. safety or liveness.

The rest of the paper is organized as follows. Sect. 2 describes brief background on the SpecC language. Sect. 3 describes the proposed approach, the synchronization verification, where the abstraction, verification and abstraction-refinement are explained in details. Then, the conclusion and outlook are given in Sect. 4.

2. SpecC Language

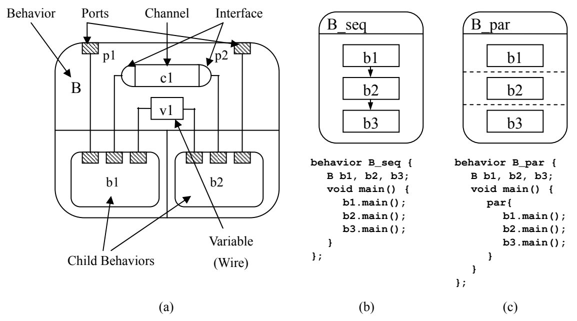
The SpecC language has been proposed as a standard system-level design language for adoption both in industry and academia. It is promoted for standardization by the SpecC Technology Open Consortium (STOC, *http://www.SpecC.org*). The SpecC language was specifically developed to address the issues involved 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.

2.1 Behavior

A behavior is a class consisting of a set of ports, a set of component instantiations and a set of private variables and functions. In order to communicate, a behavior can be connected to other behaviors or channels through its ports. Structural hierarchy can be described in SpecC as shown in Fig. 3(a). The sequential and parallel descriptions of SpecC, which will be described next, are shown in Fig. 3(b) and 3(c), respectively.

2.2 Sequentiality

Before clarifying the concurrency between statements, we have to define the semantics of sequentiality within a behav-



```
behavior B_seq {
   B b1, b2, b3;
   void main() {
      b1.main();
      b2.main();
      b3.main();
   }
};
```

```
behavior B_par {
   B b1, b2, b3;
   void main() {
      par{
         b1.main();
         b2.main();
         b3.main();
      }
   }
};
```

図 3 (a) Basic structure of SpecC model, (b) sequential description, (c) parallel description

ior. 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 Fig. 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

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

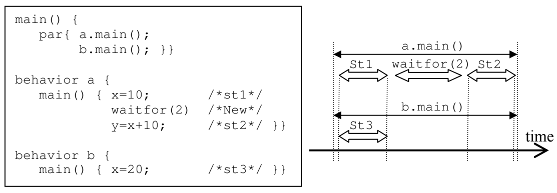
Statements in a behavior are executed sequentially but not always in continuous ways. That is, a gap may exist between Tas and $T1s$, $T1e$ and $T2s$, and $T2e$ and Tae. The lengths of these gaps are decided in non-deterministic way. Moreover, the lengths of intervals, $(T1e - T1s)$ and $(T2e - T2s)$ are non-deterministic but regarded to be close to 0 comparing with "simulation time" defined by '*waitfor*' (see [7]).

2.3 Concurrency: 'par{}' and 'notify/wait' Semantics.

Concurrency among behaviors are able to handle in SpecC with *par* and *notify/wait* semantics, as seen in Fig. 1 and 2. In a single-running of a behavior, 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 multiple 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



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

behavior a {
   main() { x=10;        /*st1*/
            waitfor(2)   /*New*/
            y=x+10;      /*st2*/ }}

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

図 4 Insertion of *waitfor* statement of Fig. 1

SpecC language, all the constraints in Fig. 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)

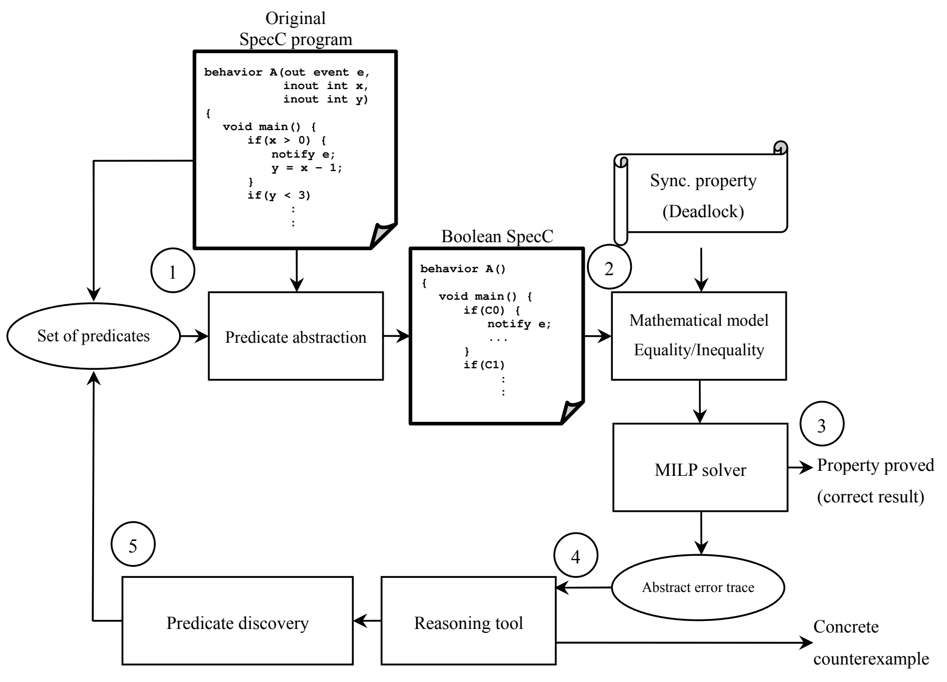
The *notify/wait* statements are used for synchronization. *wait* statements suspend their current behavior from execution and keep waiting until one of the specified events is *notify*. Let focus on the */*New*/* label in Fig. 2 of which the event manipulation statements are inserted into that of Fig. 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* due to the sequentiality in behavior *a*. Thus, it is guaranteed that *st3* is scheduled right after *st2*.

2.4 Simulation Time and 'waitfor' Semantics

With the expression *waitfor(delay)*, it implies the current behavior that executes *waitfor* statement will suspend its simulation time by 'delay' units. In order to make the semantics of sequentiality and concurrency be sound, the relationship between the length of each interval and the "simulation time" must be defined soundly. The definition is that the length of each interval on which a statement is defined is quite small and infinitely close to 0 in "simulation time". In other words, execution of each statement does not change the "simulation time". Going back to Fig. 1, this definition is intuitively described as "$(T1e - T1s)$ and $(T2e - T2s)$, the lengths of statement intervals, are infinitely close to 0". Note that this definition does allow that $(T1s - Tas)$, $(T2s - T1e)$, and/or $(Tae - T2e)$, the lengths of gaps, have non-zero value. Fig. 4 is an example where *waitfor(2)* statement is inserted between *st1* and *st2* of Fig. 1. This *waitfor(2)* increments "simulation time" by 2 units. This kind of '*waitfor*' statements are used to represent 'estimated execution time' for each statement so that performance analysis can be made in SpecC language descriptions.

3. Verification Flow

As we mentioned earlier that in both hardware and soft-



```
behavior A(out event e,
           inout int x,
           inout int y)
{
  void main() {
      if(x > 0) {
          notify e;
          y = x - 1;
      }
      if(y < 3)
          :
          :
```

```
behavior A()
{
  void main() {
      if(C0) {
          notify e;
          ...
      }
      if(C1)
          :
          :
```

図 5 The proposed verification flow

ware system design, concurrency is commonly appeared throughout design descriptions. The simple forms of synchronization and a small number of concurrent processes may be simple to verify. Unfortunately, in many practical cases, the number of processes running in parallel can be large and the synchronization will become more sophisticated.

In this paper, we propose a verification flow to check whether the given SpecC codes containing concurrent statements *par* and event manipulation statements *notify/wait* are properly synchronized. The idea of boolean programs [2] and counter-example guided abstraction refinement (CEGAR) [3] paradigm was applied for abstraction of the target SpecC code. The verification flow is shown in Fig. 5.

First, the SpecC source code must be translated into boolean SpecC code. The boolean SpecC contains only conditional (*if* or *switch*) and event manipulation statements. Second, the generated boolean SpecC is then translated into mathematical representations (equalities/inequalities) which can correctly capture the parallelism and non-deterministic of statement execution of SpecC. The property is inserted and then verify with the model checker. Verification process stops whenever the property is satisfied. Otherwise, the counter-example will be given. The following Sect. 3.1-3.5 are corresponding to each verification step of ①-⑤ as shown in Fig. 5. The corresponding pseudocodes that describe the synchronization verification is shown in Algorithm 1 and 2.

Algorithm 1 SynchronizationVerification

```
declare
  $SC$: a SpecC source code, $BS$: a boolean SpecC code
  $\tau$: a mapping of an abstraction function ($SC \xmapsto{\tau} BS$)
  $p$: a predicate, $Pre$: a set of predicates in SC
  $CE$: counterexample, $Property$: a property to verify
  $Timeout$: a threshold for limiting the computation time
begin
  unwinding loops in $SC$
  $(BS, Pre)$ := Abstraction($SC$)
  while $!Timeout \cup Pre \neq \emptyset$ do
    $(result1, CE)$ := Verify($BS, Property$)
    if $result1$ is OK then   /* property is satisfied */
      exit ("synchronization is correct")
    else
      $result2$ := ValidateCounterExample($SC, CE, Pre$)
      if $result2$ is INVALID then
        $p$ := Predicate that caused infeasibility in $ProjCE$
        $BS$ := ModifyBS($BS, p$)
        $Pre$ := $Pre - p$
      else
        exit ("synchronization is incorrect" + $CE$)
      end if
    end if
  end while
  exit ("No conclusion")
end
```

3.1 From SpecC to Boolean SpecC

The boolean programs [2] 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. We use the similar idea of the boolean programs to verify for the SpecC synchronization.

Before we abstract the SpecC descriptions to Boolean SpecC, in our verification framework, we have to unwind every loop (both finite or infinite) into a fixed finite num-

Algorithm 2 ValidateCounterExample(SC, CE, Pre)

```
declare
1:  $\tau^{-1}$: inverse of a mapping of an abstract function
2:  $ProjCE$: a projection of $CE$ to $SC$ ($CE \overset{\tau^{-1}}{\longmapsto} ProjCE$)
3:  $RenameProjCE$: a renamed path $ProjCE$
4:  $Global$: global variables appear in $ProjCE$
5:  $Race$: a race condition occurs
begin    /* $CE$ is a sequence of statements: $s_1 \ldots s_n$ */
6:  $ProjCE$ := Projection of path from $CE$ to $SC$
7:  /* Check if there is any race condition */
8:  $Race$ := CheckRaceCond($ProjCE, Global, Par$)
9:  if $Race$ is TRUE then
10:     exit ("There is a race condition")
11: end if
12: /* Renaming all assignments of each variables */
13: $RenameProjCE$ := RenameVariable($ProjCE, Par$)
14: $result2$ := Validate($RenameProjCE$)
15: return $result2$
end
```

ber of times. In other words, we convert such a loop into a fixed-length finite path. The verification results of any given property can prove the correctness of the descriptions upto the length of this finite path. This is similar to the work on bounded model checking [6].

Then, the SpecC source code is translated such that

(1) the event manipulation statements are translated into the assertion statements

(2) the conditional statements or predicates of all branching statements are automatically replaced by independent new variables, e.g. $if(x > 0)$ is replaced by $if(c0)$, $if(y < 3)$ by $if(c1)$, and so on,

(3) all those predicates are stored as a set Pre, which will be used in the refinement process (Sect. 3.5),

(4) all other statements are abstracted away by replacing with **skip** (denote in the boolean SpecC by "..." for readability).

Also, we add the property "a synchronization error of any event e occurs when $wait(e)$ was executed and $notify(e)$ was not" as an assertion to the boolean SpecC which can be done as follows:

- we consider event e in original SpecC as the logic variable in boolean SpecC,
- statement $notify(e)$ is translated to the assignment of the logic variable "*e is true*", and
- $wait(e)$ is then translated to a block of statements "*if(!(e is true)) assert(Error)*" ('*!*' is a *not* operand).

Deadlock occurs whenever $notify(e)$ has never been reached. In other words, $assert(Error)$ must have been executed since the value of e has never been triggered to be *true*. With this translations, we can verify deadlocks which may be caused by any pair of event synchronization semantics.

3.2 From Boolean SpecC to Mathematical Representations of Equalities/Inequalities

As mentioned in Sect. 2., sequentiality and concurrency are supported in SpecC. In addition, the execution of statements is non-deterministic. Hence, in order to correctly and precisely represent those characteristics of SpecC, the boolean SpecC, which has the same control flow construct as the original SpecC and contains only boolean variables, is going to be translated to mathematical representations of equalities/inequalities.

3.3 Verifying with Verification Engines

Once the mathematical representations of both the boolean SpecC and the synchronization properties are constructed, we proceed through the mixed-integer linear programming (MILP) solver to determine whether the synchronization is correct. The verification results can be either 1) *correct*, the verification process ends and the properties are proved to be correct, or 2) otherwise, the abstract counterexample is given.

This abstract counterexample needs to be validated whether it is feasible which will be described in the next section.

3.4 Validate the Abstract Counterexample

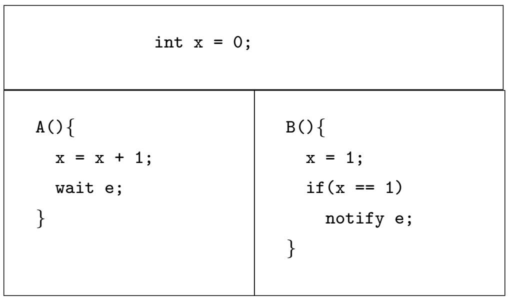
At this point, we have the abstract counterexample which contains only boolean variables. In order to validate this path, we need to refer each variable along the abstract counterexample (CE) path to the original SpecC descriptions. $ProjCE$ is the projection of CE to the original SpecC, where τ is an abstraction function from SpecC to boolean SpecC and $CE \overset{\tau^{-1}}{\longmapsto} ProjCE$. We are interesting in validating this path for its feasibility.

3.4.1 Check for Race Condition

We need to check beforehand for any race condition that might occur since it can cause the wrong verification results. Let us consider the following example in Fig. 6 where A and B are running in parallel. The global variable x is used in both A and B. Deadlock will occur whenever $x \neq 1$. It seems that $notify\ e$ is reachable. However, there is a case where deadlock can occur. That is when $x = x + 1$ was executed right after $x = 1$ which results in x equals to 2. It is obviously seen that the race condition will occur whenever there is more than one assignment of any global variable in different concurrent behaviors. The verification process terminates whenever such a race condition is found and reports which variable(s) should be re-scheduled.

3.4.2 Rename Variables

Next, before the abstract counterexample was validated, we need to rename all assignments of all variables. We can think of an assignment of a variable as generating a new vari-



```
int x = 0;
```

```
A(){
  x = x + 1;
  wait e;
}
```

```
B(){
  x = 1;
  if(x == 1)
    notify e;
}
```

図 6 A and B are running in parallel. There is a race condition on the global variable x.

able, even it is assigned to the same value, that is symbolically distinct from the one before assignment occurred. For example, $\{x = 1; if(x > 0)\ x = 2;\} \rightarrow \{x\_1 = 1; if(x\_1 > 0)\ x\_2 = 2;\}$.

Finally, we check the path $ProjCE$ which was already checked for a race condition and renamed. The validation result can be either

- path CE is *feasible* or *valid*. The verification process stops here and this path is the *counterexample* that leads to an error.
- path CE is *infeasible* (*invalid*). Maybe there is too much abstraction. This counterexample is *spurious*. The process needs to be further refined.

3. 5 Predicate Discovery & Modify Boolean SpecC

If the abstract counterexample CE was feasible in the original SpecC descriptions, then the verification process stops. The property is not hold and we now have the *real counterexample*. Otherwise, we will discover the predicate that causes this path to be infeasible. A predicate that produces a conflict in $ProjCE$, namely p, will be used for refinement of abstraction.

A predicate p that will be used for refinement can be obtained by computing the *weakest precondition* [11] of all the guard conditions along the path $ProjCE$. Once we found p that caused an error in the abstract counterexample, the next task is to obtain the modified boolean SpecC, according to predicate p, from the current boolean SpecC.

To find the location of statements that related to p, the concepts of Control-Data Flow Graph (CDFG) or Program Slicing [13] can be applied to overcome such a problem. By giving a slicing criteria (in our case, the location of p), program slicing can efficiently decompose or extract portions of program (with respect to criteria) based on control- and data-flow analysis.

4. Conclusion and Outlook

This paper presents the work of synchronization verification to cover the *automatic refinement of abstraction* which makes the entire verification, from abstraction and verification to finding counter-example and abstraction-refinement, an automatic process.

As a final remark, although this work is considered to target the verification of deadlock property. With this framework, there are several research directions to pursue in the future: synchronization verification when it comes to the complex forms of synchronization with multiple events, more efficient algorithms for the abstraction refinement, and the verification of other properties (e.g. liveness or safety).

Acknowledgements

We would like to thank the Semiconductor Technology And Research Center (STARC) organization for the support of this research.

文　献

[1] T. Ball and S. Rajamani. Boolean programs: A model and process for software analysis. Technical Report Technical Report 2000-14, Microsoft Research, Febuary 2000.

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

[3] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In E. A. Emerson and A. P. Sistla, editors, *Proceedings of the International Conference on Computer-Aided Verification (CAV'00)*, Volume 1855 of LNCS. Springer-Verlag, 2000.

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

[5] E. M. Clarke, H. Jain, and D. Kroening. Verification of specc using predicate abstraction. In *Second ACM-IEEE International Confernece on Formal Methods and Models for Codesign (MEMOCODE 2004)*, 2004.

[6] E. M. Clarke and D. Kroening. Hardware verification using ANSI-C programs as a reference. In *Proceedings of ASP-DAC 2003*, pages 308–311. IEEE Computer Society Press, January 2003.

[7] M. Fujita and H. Nakamura. The standard specc language. In *International Symposium on Systems Synthesis (ISSS 2001)*, Montreal, Canada, 2001. ACM.

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

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

[10] S. Graf and H. Saidi. Construction of abstract state graphs with pvs. In O. Grumberg, editor, *Proceeding of the International Conference on Computer Aided Verification (CAV'97)*, Volume 1254 of LNCS. Springer-Verlag, 1997.

[11] D. Gries. *The Science of Programming*. Springer-Verlag, 1981.

[12] T. A. Henzinger, R. Jhala, R. Mujumdar, and G. Sutre. Lazy abstraction. In *Proceedings of the 10th SPIN Workshop on Model Checking Software (SPIN)*, Volume 2648 of LNCS. Springer-Verlag, 2003.

[13] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. In *Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, pages 35–46. ACM Press, 1988.

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