Synchronization Verification in System-Level Design with ILP Solvers

Thanyapat Sakunkonchak Satoshi Komatsu Masahiro Fujita
VLSI Design and Education Center (VDEC), University of Tokyo
2–11–16, Hongo, Bunkyo, Tokyo, 113–0032, Japan
Phone & Fax:+(81)–3–5841–6764
E–mail: {thong,komatsu}@cad.t.u-tokyo.ac.jp, fujita@ee.t.u-tokyo.ac.jp

Abstract

Concurrency is one of the most important issues in system-level design. Interleaving among parallel processes can cause an extremely large number of different behaviors, making design and verification difficult tasks. In this work, we propose a synchronization verification method for system-level designs described in the SpecC language. Instead of modeling the design with timed FSMs and using a model checker for timed automata (such as UPPAAL or KRONOS), we formulate the timing constraints with equalities/inequalities that can be solved by integer linear programming (ILP) tools. Verification is conducted in two steps. First, similar to other software model checkers, we compute the reachability of an error state in the absence of timing constraints. Then, if a path to an error state exists, its feasibility is checked by using the ILP solver to evaluate the timing constraints along the path. This approach can drastically increase the sizes of the designs that can be verified. Abstraction and abstraction refinement techniques based on the Counterexample-Guided Abstraction Refinement (CEGAR) paradigm are applied. The proposed verification flow is introduced and some preliminary results are presented here.

1. Introduction

Building reliable hardware and software systems is a major challenge, and the system design process is made even more difficult by continual increases in design complexity. As semiconductor technology advances, entire systems can be realized within single LSIs as Systems-on-a-Chip (SoCs). At the same time, competitive pressures have been pushing system designers to shorten the design cycle and reduce time-to-market. To cope with these competing demands, new design paradigms that offer more levels of abstraction have been proposed. Designing an SoC is a process of both hardware and software development, and requires a uniform design flow from specification to implementation. Recently, there has been a lot of interest in approaches built around the C/C++ programming languages. Since C/C++ are commonly used in software development, C-based SoC design (using languages like SystemC or SpecC) is a promising approach to cover both hardware and software design with a single design/specification language.

Model checking is the formal verification technique most-commonly used in the verification of RTL or gate-level hardware designs. Due to the success of the model checking technique in the hardware domain [6, 5, 24, 7, 9], over the last few years, model checking methods have been applied to the software domain, and we have seen the birth of software model checkers for programming languages such as C/C++ and Java.

Software model checking poses its own challenges, as software tends to be less structured than hardware. In addition, concurrent software contains processes that execute asynchronously, and interleaving among these processes can cause a serious state-space explosion problem. Several techniques have been proposed to reduce the state-space explosion problem, such as partial-order reduction and abstraction. In the software verification domain, predicate abstraction [18, 2, 20] is widely applied to reduce the state-space by mapping an infinite state-space program to an abstract program of Boolean type while preserving the behaviors and control constructs of the original. Counterexample-Guided Abstraction Refinement (CEGAR) [8] is a method to automate the abstraction refinement process. More specifically, starting with a coarse level of abstraction, the given property is verified. A counterexample is given when the property does not hold. If this counterexample turns out to be spurious, the previous abstract programs are then refined to a finer level of abstraction. The verification process is continued until there is no error found or there is no solution for the given property.

Ball and Rajamani [2] propose a verification method for ANSI-C programs. It is based on the predicate abstraction and the abstraction refinement processes. A similar approach that also targets ANSI-C programs but with an on-the-fly abstraction method (lazy abstraction) is proposed in [20]. In these approaches the abstract models are verified using a BDD-based model checker or a theorem prover. SAT-based verification of ANSI-C programs is presented in [12].

In system-level design languages such as SpecC, extra constructs are added to C in order to describe the characteristics of hardware. These extra constructs support description of parallel behaviors, pipelined behaviors, finite state machines, and operations on arbitrary-length bit-vectors. System-level models are organized as a collection of cooperating processes running in parallel. In order to keep all processes executing as the designer intended, proper scheduling of statement execution in all processes (known as *synchronization*) is necessary. *Deadlock* is an error that is caused by synchronization failure.

In this work, we propose an approach to synchronization verification of systems described in SpecC. SpecC contains the *waitfor* and *notify/wait* constructs to schedule and synchronize concurrent processes. The *waitfor* statement delays a process by a specific number of time units and therefore introduces a timing constraint. While classical automata can model the transitions of a design, these transitions convey no information about the delay between two actions. It is therefore not possible to directly model a design with timing constraints. Alur and Dill [1] proposed *timed automata* as a way to incorporate quantitative information on the passage of time in automata. Model checkers for timed automata have severe constraints on their capacity, so our approach is to capture timing constraints with equalities/inequalities that can be solved by integer linear programming (ILP) tools. Verification is conducted in two steps. First, similar to other software model checkers, we compute the reachability of an error state in the absence of timing constraints. Then, if a path to an error state exists, its feasibility is checked by using the ILP solver to evaluate the timing constraints along the path. We use the CEGAR paradigm to reduce the size of the design under verification.

The paper is organized as follows. We describe some related work in Section 2. In Section 3, some background definitions of SpecC are given. In Section 4, we present the verification algorithms for the synchronization verification in SpecC. Some preliminary experimental results are presented in Section 5 and we conclude with Section 6.

2. Related Work

The study of model checking has been an active area of research during the past two decades. This extensive study led to significant new techniques, e.g. temporal logic and symbolic representations, which enabled the verification of larger and more complex systems. Model checking achieved its first industrial successes in the verification of LSI circuits and, building on these achievements, it has also been applied to the software domain.

There are two major approaches to software model checking. The first approach emphasizes *state space exploration*, where the state space of a system model is defined as the product of the state spaces of its concurrent finite-state components. The state space of a software application can be systematically explored by driving the "product" of its concurrent processes via a run-time scheduler through all states and transitions in its state space. This approach is developed in the tool Verisoft [17]. The second approach is based on *static analysis and abstraction* of software. It consists of automatically extracting a model out of a software application by statically analyzing its code and abstracting away details, and then applying symbolic model checking to verify this abstract model [13, 19, 20, 21, 29, 2, 4].

In the context of the second approach, most of the works are based on predicate abstraction [18] which conservatively transforms infinite-state systems into finite-state ones, and on the idea that of counterexample-guided abstraction refinement (CEGAR) paradigm.

The SLAM project [2, 4] conducted by Ball and Rajamani has developed a model checking tool based on the interprocedural dataflow analysis algorithm presented in [26, 27] to decide the reachability status of a statement in a Boolean program. The generation of an abstract Boolean program is expensive because it requires many calls to a theorem prover.

Clarke et al. [12] use SAT-based predicate abstraction. During the abstraction phase, instead of using theorem provers, a SAT solver is used to generate the abstract transition relation. Many theorem prover calls can potentially be replaced by a single SAT instance. Then, the abstract Boolean programs are verified with SMV. In contrast to SLAM, this work is able to handle bit-operations as well. This idea also extends to use with SpecC language [10]. The synchronization constructs *notify/wait* can be modeled, but it does not explain how to handle the timing constraints that are introduced by using *waitfor*.

Sakunkonchak and Fujita [28] propose a synchronization verification of SpecC based on the predicate abstraction of ANSI-C programs of SLAM project [2]. The idea is to mathematically model SpecC programs by equalities/inequalities formulae. Difference Decision Diagrams (DDDs) [25] are used as the verification engine, but unfortunately these cannot handle the large state space found in realistic designs.

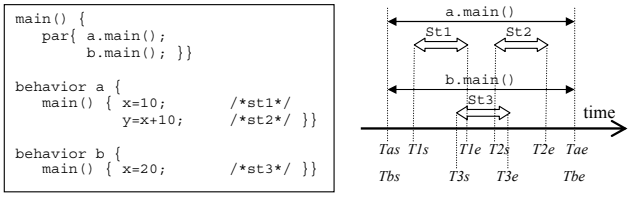

Figure 1. Timing diagram of the threads a and b under the $par\{\}$

3. The SpecC Language

The SpecC language [15, 16, 14] has been proposed as a standard system-level design language for adoption both in industry and academia. It has been 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, 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. In SpecC, the par construct allows parallel behaviors to be expressed. For example, $par\{a.main(); b.main();\}$ in Figure 1 indicates that threads a and b are running concurrently (in parallel). Within each thread, statements run in the sequential manner just as in the C programming language. The timing constraints that must be satisfied for the behavior a are $Tas \leq T1s < T1e \leq T2s < T2e \leq Tae$, where Ta, $T1$ and $T2$ stand for the timing of a, $st1$ and $st2$ respectively, and the postfix notations s and e stand for starting and ending time. In other words, $st1$ and $st2$ execute after a starts and before a ends, and no overlap is allowed in the execution of $st1$ and $st2$.

Note that it is not determined when $st3$ is scheduled relative to $st1$ and $st2$: any of "$st1 \rightarrow st2 \rightarrow st3$", "$st3 \rightarrow st1 \rightarrow st2$", and "$st1 \rightarrow st3 \rightarrow st2$" are allowed. In this case, an ambiguous result or an access violation error can occur since both $st1$ and $st3$ assign a value to the same variable x. The event manipulation statements in SpecC, $notify/wait$, can be used to synchronize threads a and b to achieve any desired scheduling. Figure 2(a) shows a modified version of Figure 1 with insertion of *notify/wait* statements. 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 enforces the scheduling "$st1 \rightarrow st2 \rightarrow st3$".

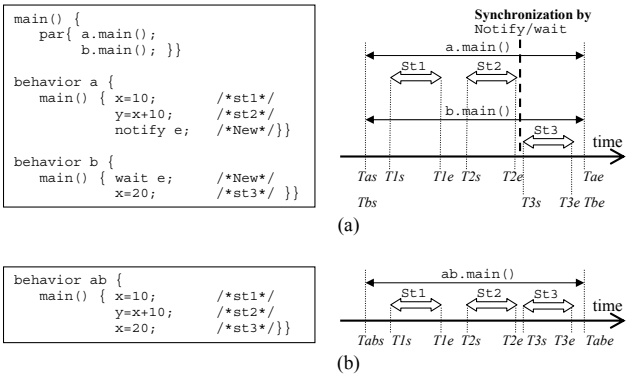

Figure 2. (a) Insertion of synchronization statement *notify/wait* of Figure 1, (b) sequential description which is equivalent to the description in (a)

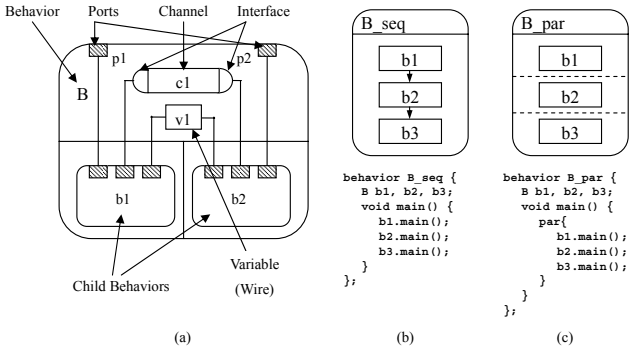

Figure 3. (a) Basic structure of SpecC model, (b) sequential description, (c) parallel description

3.1. Behaviors, Channels, and Interfaces

A SpecC 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 or interfaces. Structural hierarchy can be described in SpecC as shown in Figure 3(a). The sequential and parallel constructs of SpecC, which will be described next, are shown in Figure 3(b) and 3(c), respectively.

3.2. Sequentiality

Before clarifying the semantics of concurrency between behaviors, we have to explain sequential execution within a behavior. A behavior is defined on a time interval. Sequential statements within a behavior are also defined on time

intervals which (i) do not overlap one another and (ii) are contained in the behavior's interval. For example, in Figure 1, the beginning time and ending time of behavior a are denoted by Tas and Tae respectively, and those for $st1$ and $st2$ are $T1s$, $T1e$, $T2s$, and $T2e$. Then, the constraints that must be satisfied are

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

Statements in a behavior are executed sequentially but not necessarily contiguously. 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 a 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 [14]).

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

Concurrency and synchronization among behaviors is handled in SpecC by the $par\{\}$ and *notify/wait* constructs, as seen in Figures 1 and 2. In a single behavior running in isolation, correctness of the result is usually independent of the timing of its execution, and determined solely by the logical correctness of its functions. However, when several behaviors run in parallel, execution timing may have a great affect on the results' correctness: results can vary depending on how the multiple behaviors are interleaved. Therefore, synchronization between behaviors is an important issue for a system-level design language.

Our definition of SpecC concurrency is as follows. All behaviors invoked by the par statement have the same beginning and ending times. In Figures 1 and 2, suppose the beginning and ending time of behavior a and b are Tas and Tae and Tbs and Tbe, respectively. Then, the constraints that must be satisfied are

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

These constraints are combined with the constraints arising from sequential execution of statements within behaviors. The code in Figure 1 must therefore satisfy the following constraints:

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

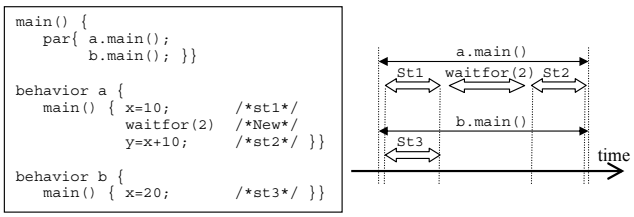

Figure 4. Insertion of *waitfor* statement of Figure 1

The *notify/wait* statements of SpecC are used for synchronization. A $wait$ statement suspends its current behavior from execution and keeps waiting until one of the specified events is notified. Let us focus on the $/*New*/$ labels in Figure 2 where the event manipulation statements are used. We can see that $wait\ e$ prevents execution of $st3$ until the event e is notified by $notify\ e$. Due to sequentiality in behavior a, *notify e* is scheduled right after the completion of $st2$. The *notify/wait* pair therefore introduces the additional constraint

$$T2e < T3s$$

Thus, it is guaranteed that $st3$ is scheduled after $st2$.

3.4. Simulation Time and 'waitfor' Semantics

The SpecC construct *waitfor(delay)* causes the behavior that executes the *waitfor* construct to suspend its simulation time by 'delay' time units. To ensure that the semantics of sequentiality and concurrency are sound, the relationship between the length of each interval and the "simulation time" must be defined soundly. We require 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". For the example of Figure 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. Figure 4 shows an example where a *waitfor(2)* statement is inserted between $st1$ and $st2$ of Figure 1. This *waitfor(2)* increments "simulation time" by 2 units and gives rise to two constraints (one for each possible interleaving of $st1$ and $st3$):

$$T1e + 2 \leq T2s,$$
$$T3e + 2 \leq T2s$$

4. Verification Framework

According to the SpecC Language Reference Manual (LRM) version 2.0, most of the execution semantics have been described using a time interval formalism, e.g. synchronization of *notify/wait* pairs or the use of simulation time for *waitfor*. In our synchronization verification framework, instead of modeling and verifying the design via timed automata, the verification flow is a collaboration between verifying the program execution by using a path simulation technique and verifying the timing constraints by using an integer linear programming solver. The verification flow is shown in Figure 5.

We are given a SpecC program and a property to verify. First, the SpecC source code is translated into Boolean SpecC code. The Boolean SpecC contains only conditional (if or $switch$) and event manipulation statements. Second, the Boolean SpecC is analyzed to obtain a set of equalities and inequalities that capture the constraints imposed by *notify/wait* and *waitfor*. The property is then verified against the Boolean SpecC program. If the property is satisfied, the verification process stops; otherwise a counterexample is given. The following Sections 4.1-4.5 correspond to each verification step ①-⑤ as shown in Figure 5. The pseudocodes that describe the synchronization verification are shown in Algorithm 1 and 2.

4.1. From SpecC to Boolean SpecC

The idea of Boolean programs [3] was proposed for software model checking. Boolean programs are expressive enough to capture the core control properties of programs and are amenable to model checking. We use the idea of Boolean programs to verify synchronization properties of SpecC.

Before we abstract the SpecC descriptions to Boolean SpecC in our verification framework, we unwind every loop (both finite and infinite) a fixed finite number of times. In other words, we convert each loop into a fixed-length finite sequence. The verification results of any given property can prove the correctness of the descriptions up to the length of this finite sequence. This is similar to the work on bounded model checking [11] where the method is conservative and guarantees that there is no *false positive* error.

Then, the SpecC source code is translated as follows:

1. the event manipulation statements *notify/wait* and *waitfor* are translated into 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,

Algorithm 1 SynchronizationVerification

declare

1: SC: a SpecC source code, BS: a Boolean SpecC code
2: τ: a mapping of an abstraction function ($SC \xmapsto{\tau} BS$)
3: p: a predicate, Pre: a set of predicates in SC
4: CE: counterexample, $Property$: a property to verify
5: $Timeout$: a threshold for limiting the computation time

begin

6: unwinding loops in SC
7: (BS, Pre) := Abstraction(SC)
8: **while** $!Timeout \cup Pre \neq \emptyset$ **do**
9: $(result1, CE)$:= Verify($BS, Property$)
10: **if** $result1$ is OK **then** /* property is satisfied */
11: **exit** ("synchronization is correct")
12: **else**
13: $result2$:= ValidateCounterExample(SC, CE, Pre)
14: **if** $result2$ is INVALID **then**
15: p := Predicate that caused infeasibility in $ProjCE$
16: BS := ModifyBS(BS, p)
17: $Pre := Pre - p$
18: **else**
19: **exit** ("synchronization is incorrect" + CE)
20: **end if**
21: **end if**
22: **end while**
23: **exit** ("No conclusion")

end

3. all those predicates are stored as a set Pre, which will be used in the refinement process (Section 4.5),

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 on any event e occurs when $wait(e)$ was executed and $notify(e)$ was not" as an assertion to the Boolean SpecC:

- we consider an event e in original SpecC as a variable in Boolean SpecC,

- statement $notify(e)$ is translated to an assignment of *true* to the variable corresponding to *e*, and

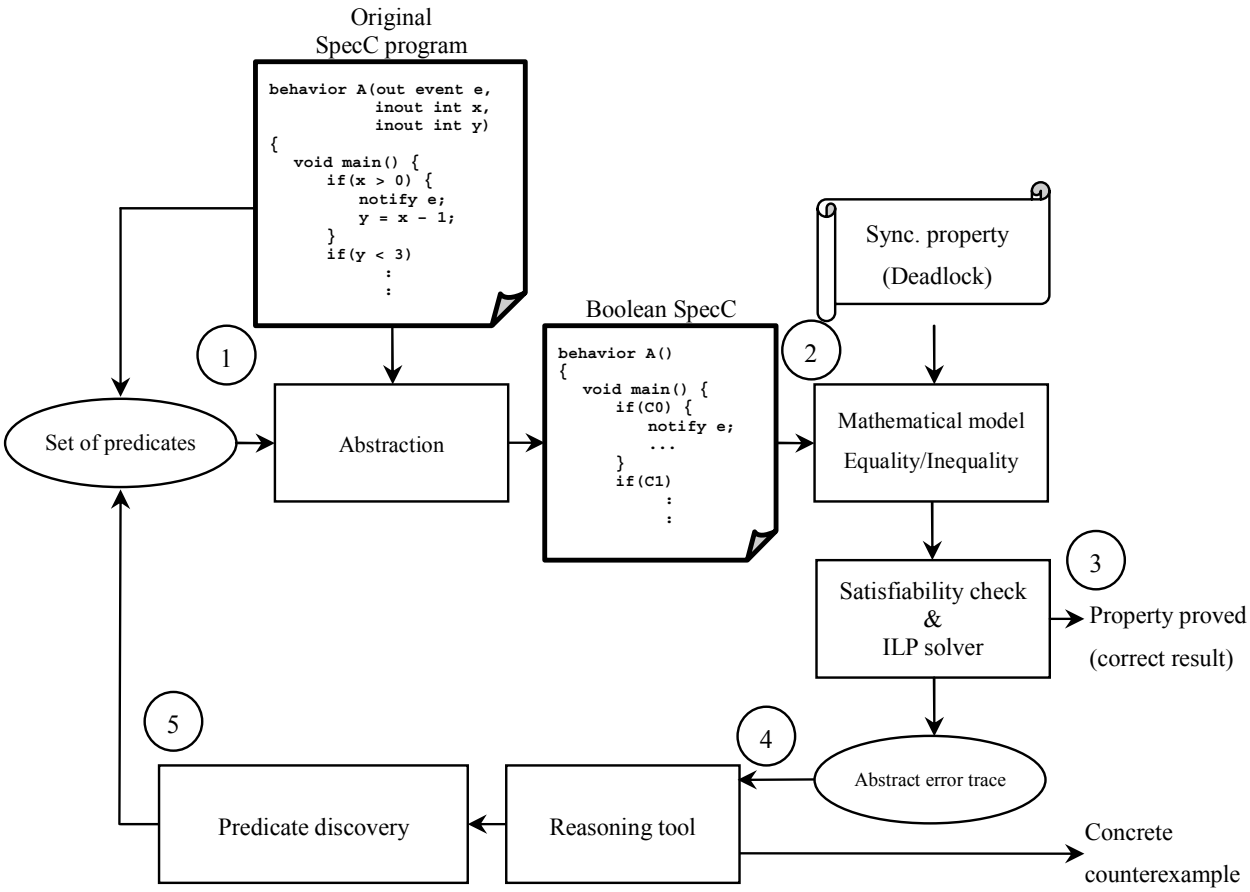

Figure 5. The proposed verification flow

- $wait(e)$ is translated to a block of statements "*if(e is NEVER true) assert(Error)*"

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

Note that in this paper, we are focusing on an automatic process for abstraction refinement of synchronization verification. We will consider the verification of synchronization of multiple events, and verification of SpecC descriptions with other properties as topics for future research.

4.2. From Boolean SpecC to Mathematical Representations of Equalities/Inequalities

As mentioned in Section 3, 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 program, which has the same control flow construct as the original SpecC and contains only Boolean variables, is translated to a mathematical representation – a set of equalities/inequalities.

4.3. Verification Engines

The property to be verified is given as an assertion statement. Checking whether a Boolean SpecC program contains an error can therefore be reduced to the problem of invariant checking (assertion violation). In other words, we check for reachability of an error state. Verification is conducted in two steps as follows.

- We check the assertion statements by reachability analysis. Since we unwound all the loops in the descriptions such that the design is now consisting of a number of directed finite paths, we can simply check the reachability by using a standard (untimed) model checker. In model checkers where the design is translated into FSMs and property can be checked based on a full reachability analysis, our method can have less

Algorithm 2 ValidateCounterExample(SC, CE, Pre)

declare

1: τ^{-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

computation. Given a synchronization property, the result can be either 1) *property holds*, there is no deadlock or 2) *property does not hold*, e.g. deadlock occurs because a *wait* statement is not notified. In the latter case an abstract counterexample is given.

- This verification step deals with synchronization of *notify/wait* and the delay of any process containing *waitfor* statements. For example, we want to find whether all *notify/wait* pairs are properly synchronized *after 20 simulation time units*. The set of equalities/inequalities arising from the Boolean SpecC and property are then solved by using ILP solver. This cannot be done in the previous step, which does not account for timing properties.

For example, consider a program with only two parallel behaviors, A and B, where behavior A contains {*waitfor(20); notify(alarm); st1;*} and behavior B contains {*wait(alarm); st2;*}. It is obvious that statement *wait(alarm)* is reachable, hence there is no deadlock error. Next, we want to check that the program is free of synchronization errors after 20 simulation time units. As described in Section 3, the descriptions of A and B can be converted into the following formulae and solved by using ILP solver.

- $TAs = TBs, TAe = TBe,$
- $TAs + 20 \leq Tst1s < Tst1e \leq TAe,$

```
int x = 0;

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

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

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

- $TBs \leq Tst2s < Tst2e \leq TBe,$
- $Tst1e < Tst2s$

If the property does not hold on the Boolean SpecC, an abstract counterexample is given. This trace is then checked for its feasibility on the original SpecC program.

4.4. Validating the Abstract Counterexample

At this point, we have an abstract counterexample that contains only Boolean variables. In order to validate this path, we need to refer each variable along the CE path to its corresponding expression in the original SpecC description. $ProjCE$ 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.

4.4.1 Checking for Race Conditions

We need to check beforehand for any race condition that might occur since races can cause incorrect results. Let us consider the example in Figure 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 $x = x + 1$ is 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.

4.4.2 Renaming Variables

Next, before the abstract counterexample is validated, we need to rename all assignments of all variables. This is to

symbolically distinguish a variable after assignment from a variable before assignment occurred. After this step each renamed variable is assigned only once. For example, $\{x = 1; if(x > 0)\ x = 2;\}$ is transformed to $\{x\_1 = 1; if(x\_1 > 0)\ x\_2 = 2;\}$.

Finally, we check the path $ProjCE$ (which is already checked for a race condition and renamed) for feasibility using the ILP solver. 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*). This counterexample is *spurious*. Maybe there is too much abstraction; the process needs to be further refined and verification reattempted.

4.5. Predicate Discovery & Boolean SpecC Refinement

If the abstract counterexample is feasible in the original SpecC description, then the verification process stops. The property does 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 the abstraction.

A predicate p that will be used for refinement can be obtained from the guarded conditions along the path $ProjCE$. Once we found a predicate p that caused an error in the abstract counterexample, the next task is to compute the modified Boolean SpecC, according to predicate p, from the current Boolean SpecC. To find the location of statements that are related to p, the concepts of Control-Data Flow Graph (CDFG) or Program Slicing [22] are used. By giving 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.

On each iteration through the refinement loop, a predicate p will be subtracted from a set of all predicates Pre. The refinement process will terminate whenever a non-spurious counterexample is found or when the set Pre is empty.

5. Preliminary Experimental Results

The SpecC descriptions used for synchronization verification were prepared such that they do not contain any of the following.

- recursive functions
- pointers
- synchronization of multiple events

Table 1. Experimental results

Benchmark	# of lines		# of		Runtime
	Original	After abs.	Behaviors	Iterations	
FIFO	260	240	5	3	18.2
Point-to-point protocol	850	500	13	2	50.1
Elevator control system	2000	800	6	2	21.1
MPEG4	48000	800	5	1	9.7

In addition, before the verification was attempted, we manually unwound each loop in the descriptions by a finite number of times. The descriptions after unwinding contain a fixed and finite-length execution path.

We inserted the conditions for verification by intentionally injecting a *wait* statement into descriptions to cause a deadlock. Then, a property was inserted to check for the error caused by that injected deadlock.

Several experiments were conducted on Pentium4 2.8GHz machine with 2GB RAM running Linux. The results of synchronization verification are shown in Table 1. A counterexample was generated whenever a property did not hold. This counterexample showed a path leading to each inserted deadlock in the descriptions. The column "# of Iterations" denotes the number of times the CEGAR refinement loop was executed. There were some properties, which we did not report here, that could not be verified using our tool. This may be because the way we handled the abstraction and refinement of predicates was not efficient. An efficient method for applying symbolic methods to predicate abstraction was recently reported in [23].

According to the results as seen in Table 1, the verification of MPEG4 descriptions considered only a portion of the descriptions (about 800 lines) instead of the entire description (about 48,000 lines). We would like to point out that focusing on the synchronization verification can significantly reduce the size of the model that needs to be considered. We also believe that once the synchronization correctness is guaranteed, we can also use this framework to verify other properties.

6. Conclusion and Outlook

Due to advances in technology, system-level design methodologies have been utilized in response to time-to-market pressures. Many tools support formal verification in then hardware and software domains, but there is little support for system-level design languages such as SpecC. We present an algorithm for formal synchronization verification of SpecC descriptions. Real-time concurrent asynchronous

systems modeled with SpecC can be verified. The SpecC descriptions are translated into equalities/inequalities and verified using an ILP solver. With this interpretation, we can check a property with respect to timing constraints. Predicate abstraction and counterexample guided abstraction refinement methods are used to abstract and refine the SpecC descriptions.

In this work, we present only verification of synchronization properties. It is simple and easy to verify other properties, e.g. safety or liveness, as long as the properties can be written as assertion statements. Future research will investigate, once all the synchronization of SpecC descriptions are guaranteed, how to do *equivalence checking* between sequential and parallel descriptions as shown in Figure 7. This is one of the important issues in system-level design methodology and we are planning to work on this direction as well.

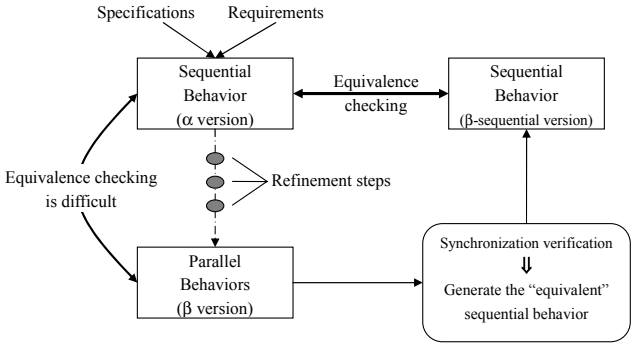

Figure 7. Synchronization verification and equivalence checking

Acknowledgment

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

References

[1] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2), April 1994.

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

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

[4] T. Ball and S. K. Rajamani. The slam toolkit. In *Proceedings of the International Conference on Computer-Aided Verification (CAV'01)*, Paris, 2001.

[5] J. R. Burch, E. M. Clarke, D. E. Long, K. L. McMillan, and D. L. Long. Symbolic model checking for sequential circuit verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, 13(4), April 1993.

[6] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2), 1992.

[7] E. M. Clarke, O. Grumberg, and D. E. Dill. Model checking and abstraction. *ACM Transactions on Programming Languages and System (TOPLAS)*, 16(5), September 1994.

[8] 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.

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

[10] 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.

[11] 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.

[12] E. M. Clarke, D. Kroening, N. Sharygina, and K. Yorav. Predicate abstraction of ansi-c programs using sat. In *Proceeding of the Model Checking for Dependable Software-Intensive Systems Workshop*, San-Francisco, USA, 2003.

[13] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Pasareanu, Robby, and H. Zheng. Bandera: Extracting finite-state models from java source code. In *Proceedings of the 22nd International Conference on Software Engineering (ICSE 2000)*, 2000.

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

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

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

[17] P. Godefroid. Model checking for programming languages using verisoft. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages*, Paris, 1997.

[18] 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.

[19] T. A. Henzinger, R. Jhala, R. Mujumdar, and G. Sutre. Lazy abstraction. In *ACM SIGPLAN-SIGACT Conference on Principles of Programming Languages*, 2002.

[20] 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.

[21] G. J. Holzmann and M. H. Smith. A practical method for verifying event-driven software. In *Proceedings of the 21st International Conference on Software Engineering (ICSE 1999)*, 1999.

[22] 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.

[23] S. K. Lahiri, R. E. Bryant, and B. Cook. A symbolic approach to predicate abstraction. In *Proceeding of the International Conference on Computer Aided Verification (CAV'03)*, Volume 2725 of LNCS, Colorado, USA, 2003. Springer-Verlag.

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

[25] J. Møller, J. Lichtenberg, H. R. Andersen, and H. Hulgaard. Difference decision diagrams. Technical Report IT-TR-1999-023, Department of Information Technology, Technical University of Denmark, Building 344, DK-2800 Lyngby, Denmark, Febuary 1999.

[26] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Principles of Programming Languages (POPL'95)*, 1995.

[27] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis with applications to constant propagation. *Theoretical Computer Science*, 167, 1996.

[28] T. Sakunkonchak and M. Fujita. Verification of synchronization in specc description with the use of difference decision diagrams. In *Forum on specification & Design Languages (FDL'02)*, Marseille, France, 2002.

[29] W. Visser, K. Havelund, G. Brat, and S. Park. Model checking programs. In *Proceedings of the 15th International Conference on Automated Software Engineering (ASE'2000)*, Grenoble, 2000.