Verification of SpecC using Predicate Abstraction*

Himanshu Jain Daniel Kroening Edmund Clarke

Carnegie Mellon University
{hjain,kroening,emc}@cs.cmu.edu

Abstract

Languages such as SystemC or SpecC offer a new design paradigm that addresses the industry's need for a fast time-to-market. However, formal verification techniques are widely applied in the hardware design industry only for low level designs, such as a netlist or RTL. The higher abstraction levels offered by these new languages are not yet amenable to rigorous, formal verification. This paper describes how to apply predicate abstraction to SpecC system descriptions. The technique supports the concurrency constructs offered by SpecC. It models the bit-vector semantics of the language accurately, and can be used for both property checking and for checking refinement together with a traditional low-level design given in Verilog.

1. Introduction

Formal verification techniques are widely applied in the hardware design industry. Introduced in 1981, *Model Checking* [10, 13] is one of the most commonly used formal verification technique in a commercial setting. However, it suffers from the state explosion problem. In case of BDD-based symbolic model checking this problem manifests itself in the form of unmanageably large BDDs [6]. This problem is partly addressed by a formal verification technique called *Bounded Model Checking* (BMC) [5]. In BMC, the transition relation for a complex design and its specification are jointly unwound to obtain a Boolean formula, which is then checked for satisfiability by using a SAT procedure such as Chaff [24]. BMC has been used successfully to find subtle errors in very large industrial circuits.

Most model-checkers used in the hardware industry use a very low level design, usually a netlist, but time-to-market requirements have rushed the Electronic Design and Automation (EDA) industry towards design paradigms that offer a very high level of abstraction. This high level can shorten the design time by hiding implementation details and by merging design layers. As part of this process, an abundance of C-like system design languages has emerged. They promise to allow joint modeling of both the hardware and software component of a system using a language that is well-known to engineers.

Several different projects have undertaken the task of extending C to support hardware specification. HardwareC [20] from Stanford University is one of the earliest C-like hardware description languages. While not all ANSI-C constructs are offered, it provides arbitrary-length bit-vector data types and an extended set of bit-vector operators. It also features inter-process communication by means of channels. It is aimed at a rather low hardware-level, resembling synthesizable RTL.

The SpecC language [1], developed at the University of California, Irvine, is based on ANSI-C and adds constructs for state machines, concurrency (pipelines in particular), and arbitrary-length bit-vectors. It also provides a way to modularize the design by a construct that resembles classes as offered by C++. Channels are used for synchronization and communication between modules.

Handel-C [26], developed at Oxford University, is very similar to SpecC, including the syntax for the extensions. As SpecC, it offers concurrency, arbitrary-length bit-vectors, and channels.

The languages mentioned above are all based on ANSI-C and share most features. On the other hand, SystemC [29], promoted by several companies in the EDA industry, is based on C++. Like the C-based languages, SystemC offers extensions to allow arbitrary-length bit-vectors and constructs for modularization and inter-process communication. As a distinguishing feature, it offers four state logic

*This research was sponsored by the Gigascale Systems Research Center (GSRC) under contract no. 9278-1-1010315, the National Science Foundation (NSF) under grant no. CCR-9803774, the Office of Naval Research (ONR), the Naval Research Laboratory (NRL) under contract no. N00014-01-1-0796, the Army Research Office (ARO) under contract no. DAAD19-01-1-0485, and the General Motors Collaborative Research Lab at CMU. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of GSRC, NSF, ONR, NRL, ARO, GM, or the U.S. government.

signals as found in Verilog. It also supports low-level hardware concepts such as multiple drivers for a single signal.

Some fragments of these languages are synthesizable, and thus allow the application of netlist or RTL-based formal verification tools. However, the higher abstraction levels offered by most of these languages are not yet amenable to rigorous, formal verification. This is caused by the high degree of asynchronous concurrency used by the models, which requires thread interleaving semantics. As languages like SpecC are closer to concurrent software than to a traditional hardware description, we propose to address this verification problem using techniques from software verification.

The effectiveness of model checking for software is severely constrained by the state space explosion problem, and much of the research in this area is targeted at reducing the state space of the model used for verification. One principal method in state space reduction of software systems is *abstraction*. Abstraction techniques reduce the program state space by mapping the set of states of the actual system to an abstract, and smaller, set of states in a way that preserves the actual behaviors of the system. If the abstraction turns out to be too coarse, it has to be refined.

The abstraction refinement process has been automated by the *Counterexample Guided Abstraction Refinement* paradigm [21, 8, 2], or CEGAR for short. One starts with a coarse abstraction, and if it is found that an error-trace reported by the model checker is not realistic, the error trace is used to refine the abstract program, and the process proceeds until no spurious error traces can be found.

Predicate abstraction of ANSI-C programs in combination with counterexample guided abstraction refinement was introduced by Ball and Rajamani [2] and promoted by the success of the SLAM project [3]. The goal of this project is to verify that Windows device drivers obey API conventions. The abstraction of the program is computed using a theorem prover such as Simplify [14], and thus, SLAM models the program variables using unbounded integer numbers. Overflow or bit-wise operators are not modeled. As the property of interest mainly depends on the control flow and not on the data computed, this treatment is sufficient.

While the original work covers sequential programs only, the idea was extended to concurrent programs in [7]. The threads are abstracted to labeled transition systems that communicate using shared events. As in the SLAM project, the abstraction is performed assuming unbounded integer numbers.

However, SystemC, SpecC and Handel C all offer an extensive set of bit-wise operators, which are not supported by this approach. At the system-level, the use of these bit-level constructs is ubiquitous. Furthermore, the languages allow the use of shared variables for communication between processes. This is not supported efficiently by the approach presented in [7].

An algorithm that preserves the bit-vector semantics during predicate abstraction is presented in [12]: A SAT solver is used to compute an abstraction of an ANSI-C program. The approach support all ANSI-C integer operators, including the bit-wise operators. The technique is described for sequential programs only, while languages like SpecC encourage the use of concurrency.

Contribution This paper describes how to use SAT-based predicate abstraction as introduced in [12] to verify a concurrent SpecC system description. Each thread of control is abstracted separately. The abstractions preserve the bit-vector semantics of SpecC, and all SpecC bit-vector operators are supported. Optionally, a low-level design (circuit level) may be added, which is also abstracted using SAT-based predicate abstraction. The abstractions of the individual threads and the optional low-level design are then composed and checked using conventional BDD-based symbolic model checking. The paper also describes the simulation and abstraction refinement process.

The low-level design may be used for two purposes:

1. The low-level design can be used to check refinement, i.e., that both the low-level and the high-level design implement the same behavior.

2. The low-level design can be used as an addition to the high-level design. The algorithm can then check safety properties on this combination. The low-level design can represent the hardware, while the high-level design represents the software component of a system.

Related Work To the best of our knowledge, this work is the first to apply predicate abstraction to SpecC or any similar system-level language.

There are tools that take a C program in a specific form as input and translate it into a circuit. The circuit can then be used for property checking or can be compared to other circuits using standard equivalence checkers, as done by Séméria et al. [30]. However, the C program has to be very similar to the circuit, e.g., they must share the same registers and must perform the computations in the same number of steps. Thus, it cannot be a high-level model such as we examine.

Matsumoto, Saito, and Fujita compare two SpecC hardware descriptions [23]. First, the differences are identified syntactically, and then compared using symbolic simulation. The method also assumes very strong similarity of the two descriptions. No abstraction is performed.

In [18], Bounded Model Checking (BMC) [5, 4] is applied to both a circuit and an ANSI-C program. The approach is restricted to sequential ANSI-C programs, no sup-

port for concurrency is provided. Furthermore, no attempt is made to abstract the program or the circuit, which limits the capacity of the method. Also, Bounded Model Checking only shows the absence of inconsistencies up to a given bound. In order to guarantee the absence of any inconsistencies, the bound has to be larger than the Completeness Threshold [19], which is too large for many industrial designs.

The concept of verifying the equivalence of a software implementation and a synchronous transition system was introduced by Pnueli, Siegel, and Shtrichman [27]. Since the target code is generated automatically by a compiler, the C program is assumed to have a specific form.

Clarke et al. [11] use SAT-based predicate abstraction for the verification of control intensive systems arising from the hardware domain. They propose a lazy abstraction refinement algorithm to identify the predicates relevant to the verification of the given property. In contrast to our work, very low level designs in the form of netlists are verified.

In [17], an algorithm for model checking safety properties of concurrent software was applied for automatic race detection in multithreaded C programs. However, their analysis does not cover hardware-like bit-vector manipulation. Qadeer et al. [28] present an algorithm for computing summaries of procedures for multi-threaded programs. The summary of a procedure P represents the effect of P on a particular input state. If P is called from two different places but with the same input state, the work done in analyzing the first call is reused for the second. They also present a model checking algorithm that uses the summaries. However, no experimental evaluation was given in the paper.

Outline In section 2, we provide a background on SpecC, and describe how we prepare the SpecC program for verification. Section 3 formalizes the semantics of the synchronization constructs. We describe the abstraction and refinement process in section 4, and provide experimental data in section 5.

2. SpecC

2.1 Introduction

The SpecC language [15] is a modeling language for the specification and design of digital embedded systems at the system level. System-level design is a methodology for specification and design of systems that include both hardware and software components. The process of system design begins with a high-level specification which specifies the functionality as well as the performance, power, cost and other constraints of the intended design.

The SpecC language is an extension of the C programming language and is based on the ANSI-C standard. As a true superset, SpecC covers the complete set of ANSI-C constructs. In addition, SpecC supports concepts essential for the design of embedded systems, including structural hierarchy, concurrency, communication, synchronization, state transitions, exception handling, and timing.

Syntactically, a SpecC program consists of a set of `behavior`, `channel`, and `interface` declarations. A behavior is similar to a C++ class with a set of ports, a set of instantiations of child behaviors, and a set of variables and functions. A behavior can be connected to other behaviors or channels through its ports. A channel is a class that encapsulates communication and provides a method for process synchronization. An interface provides a flexible link between behaviors and channels.

SpecC extends the ANSI-C syntax with several constructs for concurrent programming with asynchronous interleaving semantincs. Since the focus of this paper is making the concurrent SpecC programs amenable to verification, we describe the informal meaning of these constructs next:

- The `par` construct specifies concurrent execution. It is used to split the current thread by starting the concurrent execution of the various child threads. The execution of the `par` construct completes when all the child-threads have terminated.
- The `wait` construct suspends the execution of the current thread until a given event occurs. If more than one event is specified, the `wait` construct follows either OR or AND semantics. The OR semantics mean that the wait construct suspends the execution of the current thread until at least one of the events occurs. The AND semantics mean that the wait construct suspends the execution of the current thread until all the given events have occured. A particular ordering is not required.
- The `notify` construct generates the events specified as arguments. The execution of all threads, which are currently waiting on these events, is resumed.
- The functions defined for a channel class have an implicit locking mechanism. Only one thread is allowed to execute the channel code of a particular instance of the channel. The lock is released while the channel waits for events. We model this implicit synchronization construct using explicit lock and unlock commands.

An event has a special type called the `event` type. Note that an event does not have a value and can be used only with certain constructs such as `wait` and `notify`.

Example 1: The SpecC program of Fig. 1 shows the use of the `wait` and `notify` constructs described above. The

```
event e;
int x;

behavior A () {
   void main() {
      x = 42;
      notify e;
   }
};

behavior B () {
   void main() {
      wait(e);
      printf("Got %d", x);
   }
};

behavior Main {
   A a();
   B b();
   int main () {
      par { a.main();
            b.main(); }
      return 0;
   }
};
```

Figure 1. A simple SpecC program *P*.

example consists of a Main behavior, behavior A, and a behavior B. The Main behavior uses the `par` construct to start concurrent execution of the `main` functions of behaviors *A* and *B*, where *A* sends data to *B* via the global variable *x*. In order to ensure that *B* reads the value of *x* only when *A* has produced it, *B* waits for the event *e* to be generated by *A*.

In the example above, the use of the synchronization constructs `wait` and `notify` ensures that for any possible interleaving of the statements in thread *A* and thread *B*, the data will transfer correctly from *A* to *B*. That is, even if *A* were to generate the event *e* before B starts waiting for *e*, B will eventually get the event sent by *A* and will read the data correctly.

Informally, the synchronization semantics described in the SpecC standard require that the events generated are collected until no active thread is available for execution. Once all the threads are either suspended due to a `wait` statement or terminated, the set of generated events is delivered to the waiting threads, activating those threads that were waiting on any of them. This is why in the example above *B* is guaranteed to receive the event send by *A*.

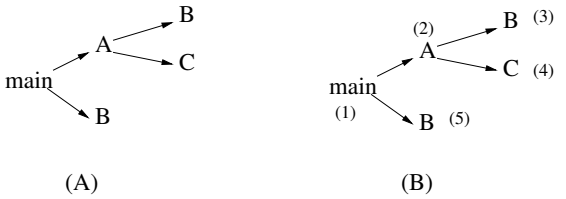

Figure 2. (A) Nested `par` structure (B) DFS numbering starting from `main`.

2.2 Pre-Processing

In this section we describe the steps used to simplify the given SpecC program. We assume that the given SpecC program does not use recursion and hence there is no dynamic thread creation either.

First, we flatten the class-like constructs offered by SpecC, i.e., the behaviors and channels. While flattening the channels, we make the implicit locks explicit by adding `lock` and `unlock` statements.

This is followed by the removal of side effects, that is, pre- and post-increment operators, the assignment operators, and the function calls. This is done by introducing temporary variables and inlining of function calls. We then replace the `break`, `continue`, `if`, `for`, `while`, and `do while` statements by equivalent guarded `goto` commands. After these steps, the program contains only guarded `goto`, assignment, `wait`, `notify`, `lock`, `unlock`, and `par` statements.

The next step is to statically create the threads that can be active during the execution of the given program. This is done by iterating over the `par` statements in the given program. For example, let the `main` thread contain a `par` statement which starts the concurrent execution of the threads of type *A* and *B*. Let *A* contain a `par` statement which starts two threads of type *B* and *C*. We assume that *B* and *C* do not contain any more `par` statements. The resulting *par graph* is shown in Fig. 2(A).

The par graph shows that there are two threads of type *B* which can be concurrent at the same time. For the static creation of the threads we need to distinguish between these two instances of *B*. This is done by performing depth first search (DFS) and assigning a distinct number called *thread-number* to each node in the par graph. The result is shown in Fig. 2(B). After assigning the thread numbers, we create five static threads, which are `main`, *A*, B_3, B_5, and *C*. The threads B_3 and B_5 are the two instances of thread *B* indexed according to their thread-numbers. We do not index the threads `main`, *A*, and *C*, because there is only one instance of these threads in Fig. 2(B).

After the creation of static threads we replace the `par`

statements using `wait` and `notify` statements. For example, consider the `main` thread of Fig. 2(B). It starts the concurrent execution of the threads A and B_5. In order to replace this `par` statement, we introduce four global events $start_1$, $start_2$, $done_1$, and $done_2$ into the system. The changes made to the code of the `main`, A, and B_5 threads are shown in Fig. 3. The `par` statement in the `main` thread is replaced by the following statements:

`notify` $start_1$, $start_2$;
`wait` $done_1$ && $done_2$;

The statements `wait` $start_1$ and `wait` $start_2$ are added to the begining of A and B_5, respectively. These statements ensure that the threads A and B_5 will wait for the `main` thread to start them by generating the events $start_1$ and $start_2$, respectively. Similarly, the statements `notify` $done_1$ and `notify` $done_2$ are added to the end of A and B_5, respectively. These events signal the `main` thread that the threads A and B_5 have completed their execution. This in turn enables the `main` thread to resume its execution.

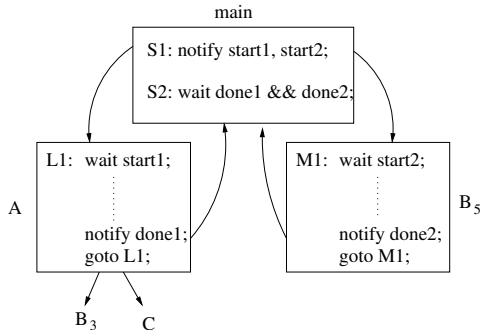

Figure 3. Replacement of `par` in the `main` thread.

The `goto` statements at the end of the threads A and B_5 in Fig. 3 cause A and B_5 to start waiting again for the events $start_1$ and $start_2$, respectively. This is required if the `par` statement of the `main` thread was inside a loop. The guards of the `goto` statements in Fig. 3 are assumed to be true.

The program obtained after applying the simplifications described above consists of a set of static threads. Each thread consists of only guarded `goto`, assignment, and the four synchronization statements.

3. Formal Semantics

The SpecC execution semantics have been described by Dömer et al. [15] using the time interval formalism. Mueller et al. [25] formalized the execution semantics of SpecC using distributed Abstract State Machines. In this section, we describe the semantics of a given SpecC program P by defining a transition system T for P. The transition system $T = (S, I, R)$ consists of a set of states S, a set of initial states $I \subseteq S$, and a transition relation $R(s, s')$, which relates the current state $s \in S$ to a next-state $s' \in S$.

We assume that the given program P has already been pre-processed as described in section 2.2. Let $\{P_1, \ldots, P_m\}$ be the set of the static threads present in P. A state s of the program P consists of the valuations for:

- the set of program counters $\{pc_1, \ldots, pc_m\}$, where each pc_i is the program counter of the thread P_i. The projection function $pc_i(s)$ maps a state s to the value of the program counter pc_i in state s.
- the set of program variables, denoted by V. The function $v(s)$ maps a state s to the value of the variable v in state s.
- the set of *event bits* E, defined as $\cup_{i=1}^{m} \cup_e \{e_i\}$, where e denotes an event in the program. Intuitively, an event bit e_i is the flag used by the thread P_i to check the occurence of event e. The function $e_i(s)$ maps s to the value of the event bit e_i in state s.

Henceforth, we assume that i, j range over the thread indexes, that is $i, j \in \{1, \ldots, m\}$.

Initially, the program counter for each thread is set to one and all event bits e_i are set to false. Thus, the set of initial states I is defined as follows:

$$I \quad := \quad \{s \in S | (\forall i.\ pc_i(s) = 1) \wedge (\forall e_i \in E.\ \neg e_i(s))\}$$

The transition relation $R(s, s')$ relates two states s and s', where s' is obtained by choosing one of the threads P_i nondeterministically and executing it in the state s. If the thread P_i is executed in the transition from s to s', then the program counters of all the other threads $j \neq i$ remain the same. We use $\delta(s, s', i)$ to denote effect of executing the thread P_i in state s. Formally,

$$R(s, s') := \exists i\ (\forall j \neq i\ \rightarrow pc_j(s) = pc_j(s')) \wedge \delta(s, s', i)$$

We use $eqvars(s, s')$ to denote that the values of all variables do not change in the transition from s to s'.

$$eqvars(s, s') \quad := \quad \forall v \in V.\ v(s) = v(s')$$

We use $eqevents(s, s')$ to denote that the values of all the event bits do not change in the transition from s to s'.

$$eqevents(s, s') \quad := \quad \forall e_i \in E.\ e_i(s) = e_i(s')$$

We assume that in each transition exactly one thread executes one statement atomically. This assumption is justified later in this section. Let $\Gamma(s, i)$ denote the statement executed in the state s by P_i. The function $\delta(s, s', i)$ is defined

by a case split on the statement $\Gamma(s,i)$. We have the following cases:

If $\Gamma(s,i)$ is a guarded `goto` statement of the form $(\texttt{goto},g,l)$, then the value of the program counter pc_i is changed according to the value of the boolean condition g in the state s, which is denoted by $g(s)$. If $g(s)$ is true, then the program counter is set to l, otherwise the program counter is simply incremented. The values of the variables and the values of the event bits remain unchanged.

$$\delta(s,s',i) := \begin{cases} pc_i(s') = l & : g(s) \\ pc_i(s') = pc_i(s)+1 & : \text{otherwise} \end{cases} \\ \wedge\ eqvars(s,s') \ \wedge\ eqevents(s,s')$$

If $\Gamma(s,i)$ is an assignment statement of the form (`v:=exp`), then the value of v is set to the value of the expression exp in the state s, which is denoted by $exp(s)$. The values of the other variables and the values of the event bits in E remain unchanged. The program counter for P_i is incremented.

$$\begin{aligned} \delta(s,s',i) \quad := \quad & (v(s') = exp(s)) \wedge \\ & (\forall u \in V \backslash \{v\} :\ u(s) = u(s')) \wedge \\ & (pc_i(s') = pc_i(s)+1) \wedge eqevents(s,s') \end{aligned}$$

If $\Gamma(s,i)$ is a wait statement of the form (`wait`,AND,W), where W is a set of events, then the thread P_i waits until all the events in W have been generated (AND semantics). In order to test if an event e has been generated, the thread P_i checks the event bit e_i. If all the event bits e_i with $e \in W$ are true, all the events in W have been generated. In this case, the program counter for P_i is incremented and the event bits e_i with $e \in W$ are reset to false. The values of the other event bits remain the same. We denote the set of other event bits by E' with $E' = E \backslash \{e_i | e \in W\}$. If not all the events in W have been generated yet, then the program counter for P_i remains unchanged. The values of all the event bits remain unchanged. In both the cases, the values of all the variables remain unchanged.

$$\begin{aligned} \bigwedge_{e \in W} e_i(s) \quad \rightarrow \quad & \delta(s,s',i) := eqvars(s,s') \wedge \\ & (pc_i(s') = pc_i(s)+1) \wedge \\ & \bigwedge_{e \in W} \neg e_i(s') \ \wedge \\ & (\forall f_j \in E' :\ f_j(s) = f_j(s')) \\ \neg \bigwedge_{e \in W} e_i(s) \quad \rightarrow \quad & \delta(s,s',i) := (pc_i(s') = pc_i(s)) \wedge \\ & eqvars(s,s') \wedge eqevents(s,s') \end{aligned}$$

The treatement of the `wait` statement with OR semantics is similar.

If $\Gamma(s,i)$ is a notify statement of the form (`notify`,W), where W is a set of events, then for every event $e \in W$, we set the event bits e_j for all j $(1 \leq j \leq m)$ to true. This ensures that any thread P_j that was previously waiting for an event $e \in W$ will now find the corresponding event bit e_j to be true. This also allows `notify` e to match with `wait` e even if `wait` e occurs later.

$$\begin{aligned} \delta(s,s',i) \quad := \quad & (\forall e \in W\ \forall j :\ e_j(s')) \wedge \\ & (\forall e \notin W\ \forall j :\ e_j(s) = e_j(s')) \wedge \\ & (pc_i(s') = pc_i(s)+1) \wedge eqvars(s,s') \end{aligned}$$

Our definition of the transition relation assumes that in each transition exactly one thread executes one statement atomically. However, the SpecC standard does not guarantee atomicity for the execution of any portion of the concurrent code. The SpecC standard requires that for concurrent threads to be cooperative, the threads need to be synchronized at the point of communication.

If the given program is not synchronized properly, the following situation might arise: thread P_1, executing the assignment statement $x := y$, is preempted by another thread P_2, which starts writing to y. As a result of this, x might get a value with bits from both the old and the new value of the variable y. This situation is commonly referred to as the *read write* (`RW`) conflict between two concurrently executing threads. A situation similar to this is the *write write* (`WW`) conflict which arises when two threads attempt to write to a shared variable simultaneously.

Both `RW` and `WW` conflicts are undesirable, as they make the program unsafe. Therefore, before taking a transition out of a state s, we first check for a potential `RW` or `WW` conflict in the state s. In order to do this, we compute for each thread P_i the set of variables it can read and write in the state s. We denote these sets by $read(i,s)$ and $write(i,s)$, respectively. The presence of a `RW` or `WW` conflict can be cast as the following safety property:

$$\begin{aligned} \exists i \exists j :\ (i \neq j) \quad \wedge \quad & ((read(s,i) \cap write(s,j) \neq \emptyset) \vee \\ & (write(s,i) \cap write(s,j) \neq \emptyset)) \end{aligned}$$

We call a state s in which a `RW` or `WW` conflict is possible during the execution of the next statement of two threads a *conflict state*. If there is a conflict state s, we report that as an error and stop the verification process. However, if there is no `RW` or `WW` conflict in s, then we can safely make a transition out of state s using the transition relation described above. This is justified by the Claim 1.

Claim 1 *Assuming that the execution is free of* `RW` *and* `WW` *conflicts, any state s reachable by executing k statements using full interleaving semantics (that is, no atomicity) is also reachable by k transitions using interleavings only between statements (that is, atomic execution of the statements).*

This claim is shown by induction on k.

Claim 2 *If there is a conflict state s reachable using full interleaving semantics, it is also reachable using interleavings only between statements.*

This claim is also shown inductively. It allows us to conclude that it is sufficient to check for possible `RW` or `WW` conflicts before the execution of a statement. It is not necessary to consider any interleavings within the statement.

4. Computing the Abstraction

4.1. Predicate Abstraction

We verify the SpecC program using counterexample guided abstraction refinement (CEGAR). We perform a predicate abstraction [16], i.e., the variables of the program are replaced by Boolean variables that correspond to a predicate on the original variables.

The first step is to obtain an initial abstraction. This abstraction is then checked using a symbolic model checker. We perform a safe abstraction, i.e., if the property holds on the abstract model, we can conclude that it also holds on the concrete model. If the property does not hold on the abstract model, we expect the model checker to provide a counterexample. This abstract counterexample is then simulated on the concrete model. This step corresponds to Bounded Model Checking on the concrete model with additional constraints that are derived from the abstract counterexample.

If the simulation is successful, we obtain a concrete counterexample from the Bounded Model Checker, which can be given to the user to aid in finding the cause of the flaw. If the simulation fails, the abstract counterexample is spurious, and the abstraction has to be refined.

Formally, we assume that the algorithm maintains a set of n predicates $p_1, \ldots, p_n$. These predicates are global, i.e., the abstract model only contains one set which is used by all the threads. The predicates are functions that map a concrete state $x \in S$ into a Boolean value. When applying all predicates to a specific concrete state, one obtains a vector of n Boolean values, which represents an abstract state $\hat{x}$. We denote this function by $\alpha(x)$. It maps a concrete state into an abstract state and is therefore called an *abstraction function*.

We perform an existential abstraction [9], i.e., the abstract model can make a transition from an abstract state $\hat{x}$ to $\hat{x}'$ iff there is a transition from x to x' in the concrete model and x is abstracted to $\hat{x}$ and x' is abstracted to $\hat{x}'$. We call the abstract product machine $\hat{T}$, and we denote the transition relation of $\hat{T}$ by $\hat{R}$.

$$\hat{R} \quad := \{(\hat{x},\hat{x}') \mid \exists x,x' \in S : \quad R(x,x') \wedge \alpha(x) = \hat{x} \wedge \alpha(x') = \hat{x}'\}$$

Note that in practice, additional transitions are often added to the abstract transition relation in order to make the computation of $\hat{R}$ easier. This is common for the abstraction of both circuits and programs.

The abstraction of a safety property $P(x)$ is defined as follows: for the property to hold on an abstract state $\hat{x}$, the property must hold on all states x that are abstracted to $\hat{x}$.

$$\hat{P}(\hat{x}) \quad :\Longleftrightarrow \quad \forall x \in S : (\alpha(x) = \hat{x}) \Longrightarrow P(x)$$

The same abstraction is also used for the initial state predicate. Thus, if P holds on all reachable states of the abstract model, P also holds on all reachable states of the concrete model.

4.2. SAT-based Abstraction

Most tools using predicate abstraction for software verification use general-purpose theorem provers such as Simplify [14] to compute the abstraction. This approach suffers from the fact that errors caused by bit-vector overflow may remain undetected. Furthermore, bit-wise operators are usually treated by means of uninterpreted functions. Thus, properties that rely on these bit-vector operators cannot be verified. However, we expect that system-level SpecC models typically use an abundance of bit-wise operators, and that the property of interest will depend on these operations.

In [12], the authors propose to use a SAT solver to compute the abstraction of a sequential ANSI-C program. This approach supports all ANSI-C integer operators, including the bit-wise operators. It is used to abstract the assignment statements and the guards of the guarded goto statements. No abstraction is done for the `wait` and `notify` statements. They are copied into the abstract model directly using the event bits (section 4.3).

Assignment Statements In order to abstract an assignment statement `v := exp`, it is transformed into an equality $v' = exp$. The primed version of a variable denotes the value of the variable in the next state. This equality is conjoined with equalities that define the next value of any other variable $u \in V \backslash \{v\}$ to be the current value. Thus, only the value of the variable v in the assignment statement changes. This equation system is denoted by $\mathcal{T}$, $\bar{v}$ denotes the vector of all variables in V.

$$\mathcal{T}(\bar{v},\bar{v}') \quad := \quad v' = exp \wedge \bigwedge_{u \in V \backslash \{v\}} u' = u$$

The abstract transition relation $\mathcal{B}(\hat{x},\hat{x}')$ relates a current state $\hat{x}$ (before the execution of the assignment) to a next state $\hat{x}'$ (after the execution of the assignment). It is defined using α as follows:

$$\{(\hat{x},\hat{x}') \mid \exists \bar{v},\bar{v}' : \ (\alpha(\bar{v}) = \hat{x}) \wedge \mathcal{T}(\bar{v},\bar{v}') \wedge (\alpha(\bar{v}') = \hat{x}')\}$$

We compute $\mathcal{B}$ using SAT-based Boolean quantification, as described in [12]. The result is DNF over the predicates.

Branching Conditions The expressions used in the branching conditions of the program are ideal candidates for predicates, and thus, the branching condition will often be a Boolean combination of predicates. If this is so, the branching conditions are simply replaced by their corresponding Boolean variables. If not, the expression is abstracted using SAT in analogy to an assignment statement.

4.3. Checking the Abstract Model

The abstraction process above results in one Boolean program for each thread. The programs share the predicates, but each thread has individual state bits to store the events. No attempt is made to abstract the event structure. We rely on the model checker to explore the possible interleavings of the individual threads. In order to check the abstract model, we use SMV.

The `wait` and the `notify` statements present in the static threads are directly translated to the SMV statements using the semantics described in section 3. For example, consider a program with only two threads P_1 and P_2. Let P_1 contain a `wait` e statement and let P_2 contain a `notify` e statement. In order to translate these statements to SMV, two event bits e_1 and e_2 are introduced into the SMV model. Let l_1 and l_2 denote the program counter values at the `wait` e statement in P_1 and the `notify` e statement in P_2, respectively. The SMV statements generated for `wait` e statement in P_1 are as follows:

```
ASSIGN next(pc1) :=
case pc1=l1:     // wait statement
     case  e1 :  l1+1;  // event e has occured
           !e1 : l1;  // event e has not yet occured
     esac
esac
```

`TRANS` $pc_1 = l_1 \wedge e_1 \rightarrow$ `!next`(e_1) // resetting e_1

The SMV statement generated for the `notify` e statement in P_2, is as follows:

`TRANS` $pc_2 = l_2 \rightarrow$ `next`$(e_1) \wedge$ `next`(e_2)

As described in section 3, it is not necessary to consider all possible interleavings if one checks for possible conflicts before the execution of the statements. We merge multiple assignment statements into one basic block and abstract this block into one abstract transition, and thus, we eliminate the interleavings within a basic block. This requires that any conflict between any pair of statements in the basic blocks that are about to be executed has to be detected. The set of variables read and written until the end of the basic block can easily be computed statically. We use these sets to detect a potential `RW` or `WW` conflict among the threads that are ready to be executed by means of an SMV `SPEC` statement.

4.4. Simulation and Refinement

If the property does not hold on the abstract model, SMV returns a counterexample trace. This trace is then checked on the concrete model.

Let the counterexample trace have k steps. Each step is performed by a particular thread, and corresponds to a particular statement in the concrete program. We use the thread schedule (interleaving) of the abstract trace as given by SMV for the simulation. No attempt is made to find alternate thread schedules.

The simulation requires a total of k SAT instances. Each instance adds constraints for one more step of the counterexample trace. We denote the value of the (concrete) variable $v \in V$ after step i by v_i. All the variables $v \in V$ inside an arbitrary expression e are renamed to v_i using the function $\rho_i(e)$.

The SAT instance number i is denoted by Σ_i and is built inductively as follows: Σ_0 (for the empty trace) is defined to be true. For $i \geq 1$, Σ_i depends on the type of statement of state i in the counterexample trace. Let p_i denote the statement executed in the step i.

If step i is a guarded goto statement, then the (concrete) guard g of the goto statement is renamed and used as conjunct. Furthermore, a conjunct is added that constraints the values of the variables to be equal to the previous values:

$$p_i = (\texttt{goto}, g, l) \quad \longrightarrow \quad \Sigma_i := \Sigma_{i-1} \wedge \rho_i(g) \wedge \bigwedge_{u \in V} u_i = u_{i-1}$$

If step i is an assignment statement, the equality for the assignment statement is renamed and used as conjunct:

$$p_i = (\texttt{v:=exp}) \quad \longrightarrow \quad \Sigma_i := \Sigma_{i-1} \wedge \rho_i(v) = \rho_{i-1}(exp) \wedge \bigwedge_{u \in V \setminus \{v\}} u_i = u_{i-1}$$

If step i is a `notify` or `wait` statement, the variables are not changed.

$$p_i = (\texttt{notify}, W) \quad \longrightarrow \quad \Sigma_i := \Sigma_{i-1} \wedge \bigwedge_{u \in V} u_i = u_{i-1}$$

The formal definition of Σ_i for `wait` statements is done analogously.

Note that in case of assignment, `wait`, and `notify` statements, Σ_i is satisfiable if the previous instance Σ_{i-1} is satisfiable. Thus, the check only has to be performed if the last statement is a guarded goto statement. If the last instance Σ_k is satisfiable, the simulation is successful and a bug is reported. The satisfying assignment provided by the

SAT solver allows us to extract the values of all variables along the trace. If any SAT instance is unsatisfiable, the step number and the guard that caused the failure are passed to the refinement algorithm.

Refinement If the abstract counterexample cannot be simulated, it is an artifact from the abstraction process and the abstraction has to be refined. This is done by computing the weakest precondition of the guard g that caused the last SAT-instance Σ to be unsatisfiable. The weakest preconditions are computed following the simulation trace as built in the previous section, and thus, the computation may include statements from multiple threads. The new predicates obtained from these weakest pre-conditions are added to the global set of predicates. This ensures that in future abstractions, this particular spurious counterexample will not occur.

5. Experimental Results

We report experimental results for synthetic benchmarks to evaluate the scalability of the approach with respect to the size of the program, the number of threads, and the number of predicates required to prove or disprove the property. The experiments are performed on a 1.5 GHz AMD machine with 3 GB of memory running Linux.

The benchmark results are given in table 1. The `PIPE` benchmarks is a series of instances of a pipeline that simply passes data through. The number denotes the number of pipeline stages. Each pipeline stage is modeled as a separate thread. A separate event for each stage is used to synchronize the communication of the threads. The property used asserts that the data that was put in the pipeline matches the data that comes out of the pipeline. The runtime includes the time for the abstraction refinement. The table shows the total time and the time spent in the model checker checking the abstract model. On this benchmark, the run-time is clearly dominated by the time required for checking the abstract model. Thus, we experimented with two different implementations, CMU SMV and NuSMV. NuSMV clearly outperforms CMU SMV, and therefore we only report the NuSMV time. Both model checkers show exponential runtime in the number of threads.

The `PRED n` benchmarks require n predicates and refinement iterations to show the property. While the abstraction scales well with the number of predicates, the model checker quickly becomes the bottleneck.

The `ALUPIPE` benchmarks use a SpecC program that models a shallow pipeline (just two or three stages). However, they make extensive use of bit-wise operators (arithmetic, slicing, concatenation). E.g., the program computes the result of an addition in multiple steps. The property is an assertion that checks the result computed by the pipeline. These benchmarks require many predicates, and thus, the run-time is dominated by the abstraction phase.

6. Conclusion

An abundance of formal verification tools are available for the verification of hardware given in RTL or as a netlist. However, there is little support for formal verification for system level languages such as SpecC. We presented an algorithm for rigorous, formal verification of SpecC programs. The algorithm models the bit-vector semantics of the language accurately, and provides full support for the concurrency and synchronization constructs offered by the language.

The method uses counterexample guided abstraction refinement to obtain a safe predicate abstraction of the SpecC program. The abstraction is done using SAT, which enables support for all bit-vector operators. The experimental results indicate that the verification of the abstract model can be a bottleneck if many threads are used. Future research will investigate the use of partial order reduction on these abstract models [22]. We are also investigating the use of explicit state and SAT-based model checkers.

Acknowledgement

We thank Masahiro Fujita for numerous clarifications of the semantics of SpecC.

References

[1] http://www.specc.org.

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

[3] T. Ball and S. K. Rajamani. Automatically validating temporal safety properties of interfaces. In *The 8th International SPIN Workshop on Model Checking of Software*, volume 2057 of *LNCS*, pages 103–122. Springer, 2001.

[4] A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, and Y. Zhu. Symbolic model checking using SAT procedures instead of BDDs. In *Design Automation Conference (DAC'99)*, 1999.

[5] A. Biere, A. Cimatti, E. M. Clarke, and Y. Yhu. Symbolic model checking without BDDs. In *Tools and Algorithms for Construction and Analysis of Systems*, pages 193–207, 1999.

[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):142–170, 1992.

[7] S. Chaki, E. Clarke, A. Groce, J. Ouaknine, O. Strichman, and K. Yorav. Efficient verification of sequential and concurrent C programs. *Formal Methods in System Design (FMSD)*, 2004. to appear.

Benchmark	threads	bug length	predi-cates	Runtime	
				Total	NuSMV
PIPE 4	5	-	4	1.9	1.9
PIPE 5	6	-	5	4.4	4.3
PIPE 6	7	-	6	8.3	8.2
PIPE 7	8	-	7	13.2	13.1
PIPE 8	9	-	8	23.3	23.2
PIPE 9	10	-	9	32.6	32.5
PIPE 10	11	-	10	55.9	55.7
PIPE 11	12	-	11	75.8	75.6
PIPE 12	13	-	12	92.0	91.9
PIPE 13	14	-	13	202.3	202.1
PIPE 14	15	-	14	789.2	788.9

Benchmark	threads	bug length	predi-cates	Runtime	
				Total	NuSMV
PRED 8	1	-	8	0.9	0.6
PRED 16	1	-	16	6.6	4.5
PRED 32	1	-	32	60.3	46.4
PRED 64	1	-	64	831.6	723.1
ALUPIPE A	3	-	4	4.0	0.3
ALUPIPE B	3	25	1	2.8	0.3
ALUPIPE C	3	-	6	11.1	2.6

Table 1. Experimental Results. The times are given in seconds. The "bug length" column denotes the length of the counterexample. A dash denotes that the property holds.

[8] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and V. H. Counterexample-guided abstraction refinement. In *CAV*, pages 154–169. Springer-Verlag, 2000.

[9] E. Clarke, O. Grumberg, and D. Long. Model checking and abstraction. In *POPL*, 1992.

[10] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[11] E. Clarke, O. Grumberg, M. Talupur, and D. Wang. High level verification of control intensive systems using predicate abstraction. In *First ACM and IEEE International Conference on Formal Methods and Models for Co-Design (MEMOCODE03)*. IEEE, 2003.

[12] E. Clarke, D. Kroening, N. Sharygina, and K. Yorav. Predicate abstraction of ANSI-C programs using SAT. In *Proc. of the Model Checking for Dependable Software-Intensive Systems Workshop, San-Francisco, USA*, 2003.

[13] E. M. Clarke and E. A. Emerson. Synthesis of synchronization skeletons for branching time temporal logic. In *Logic of Programs: Workshop*, volume 131 of *LNCS*. Springer-Verlag, 1981.

[14] D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: A theorem prover for program checking. Technical Report HPL-2003-148, HP Labs, 2003.

[15] R. Dömer, J. Zhu, and D. D. Gajski. The SpecC language reference manual version 2.0.

[16] S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In O. Grumberg, editor, *Proc. 9th INternational Conference on Computer Aided Verification (CAV'97)*, volume 1254, pages 72–83. Springer Verlag, 1997.

[17] T. A. Henzinger, R. Jhala, R. Majumdar, and S. Qadeer. Thread-modular abstraction refinement. In *Proceedings of the 15th International Conference on Computer-Aided Verification (CAV)*, pages 262–274. Lecture Notes in Computer Science 2725, Springer-Verlag, 2003.

[18] D. Kroening, E. Clarke, and K. Yorav. Behavioral consistency of C and Verilog programs using bounded model checking. In *Proceedings of DAC 2003*, pages 368–371. ACM Press, 2003.

[19] D. Kroening and O. Strichman. Efficient computation of recurrence diameters. In L. Zuck, P. Attie, A. Cortesi, and S. Mukhopadhyay, editors, *4th International Conference on Verification, Model Checking, and Abstract Interpretation*, volume 2575 of *Lecture Notes in Computer Science*, pages 298–309. Springer Verlag, January 2003.

[20] D. Ku and G. DeMicheli. HardwareC – a language for hardware design (version 2.0). Technical Report CSL-TR-90-419, Stanford University, 1990.

[21] R. Kurshan. *Computer-aided verification of coordinating processes: the automata-theoretic approach*. Princeton University Press, 1994.

[22] F. Lerda, N. Sinha, and M. Theobald. Symbolic model checking of software. In B. Cook, S. Stoller, and W. Visser, editors, *Electronic Notes in Theoretical Computer Science*, volume 89. Elsevier, 2003.

[23] T. Matsumoto, H. Saito, and M. Fujita. Equivalence checking of c-based hardware descriptions by using symbolic simulation and program slicer. In *International Workshop on Logic and Synthesis (IWLS'03)*, 2003.

[24] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference (DAC'01)*, pages 530–535, June 2001.

[25] W. Mueller, R. Dmer, and A. Gerstlauer. The formal execution semantics of SpecC. In *Proc. of International Symposium on System Synthesis*, 2002.

[26] I. Page. Constructing Hardware-Software Systems from a Single Description. *Journal of VLSI Signal Processing*, 12(1):87–107, 1996.

[27] A. Pnueli, M. Siegel, and O. Shtrichman. The code validation tool (CVT)- automatic verification of a compilation process. *Int. Journal of Software Tools for Technology Transfer (STTT)*, 2(2):192–201, 1998.

[28] S. Qadeer, S. K. Rajamani, and J. Rehof. Summarizing procedures in concurrent programs. In *31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 245–255, 2004.

[29] http://www.systemc.org.

[30] L. Séméria, A. Seawright, R. Mehra, D. Ng, A. Ekanayake, and B. Pangrle. RTL C-based methodology for designing and verifying a multi-threaded processor. In *Proc. of the 39th DAC*, pages 123–128. ACM Press, 2002.