Form Method Syst Des (2006) 30:5–28
DOI 10.1007/s10703-006-0020-3

Verification of SpecC using predicate abstraction

Edmund Clarke · Himanshu Jain · Daniel Kroening

Published online: 31 August 2006

Abstract Languages such as SystemC or SpecC offer modeling of hardware and whole system designs at a high level of abstraction. 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 both for property checking and for checking refinement together with a traditional low-level design given in Verilog.

Keywords Verification · System level design · Predicate abstraction

This paper is an extended version of [29].

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.

E. Clarke (✉) · H. Jain
Computer Science Department, Carnegie Mellon University, Pittsburgh, PA, USA
e-mail: emc@cs.cmu.edu

H. Jain
e-mail: hjain@cs.cmu.edu

D. Kroening
Computer Systems Institute, ETH Zurich, Switzerland
e-mail: daniel.kroening@inf.ethz.ch

1 Introduction

Formal verification techniques are widely applied in the hardware design industry. Introduced in 1981 *Model Checking* [11, 15] is one of the most commonly used formal verification techniques in a commercial setting. However, it suffers from the state explosion problem. In the case of BDD-based symbolic model checking this problem manifests itself in the form of unmanageably large BDDs [8]. This problem is partly addressed by a formal verification technique called *Bounded Model Checking* (BMC) [7]. 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 [40]. BMC has been used successfully to find subtle errors in very large industrial circuits [20, 48].

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 allowing the creation of fast executable verification models. This way, bugs in the design can be discovered early in the design process. As part of this paradigm, an abundance of C-like system design languages has emerged. They promise 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 [36] 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 (an array of bits) 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.

Handel-C [42], developed at Oxford University, is based on ANSI-C and adds concurrency primitives, arbitrary-length bit-vectors, and channels. The synchronization mechanisms are derived from the semantics of CSP.

The SpecC language [23], developed at the University of California, Irvine, is also 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.

The languages mentioned above are all based on ANSI-C and share most features. On the other hand, SystemC [1], 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 signals as found in Verilog. A four state logic signal can have the four logical values '0', '1', 'X' (zero or one), and 'Z' (high impedance). SystemC 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. The ambiguity in the specifications of the underlying programming languages such as C and C++ makes the creation of formal models for verification even more difficult. 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 [4, 16, 37], 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 [25] is a powerful technique for extracting finite-state models from complex source code. It abstracts data by keeping track of certain predicates on the data. Each predicate is represented by a Boolean variable in the abstract program, while the original data variables are eliminated. Predicate abstraction of ANSI-C programs in combination with counterexample guided abstraction refinement was introduced by Ball and Rajamani [4] and promoted by the success of the SLAM project [5]. The goal of SLAM is to verify that Windows device drivers obey API conventions. The abstraction of the program is computed using a theorem prover such as Simplify [21], 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 on predicate abstraction covers sequential programs only, the idea was extended to concurrent programs in [9]. The threads are abstracted to labeled transition systems that communicate using shared events. However, system-level languages allow the use of shared variables for communication between processes. This is not supported efficiently by the approach presented in [9]. As in the SLAM project, the abstraction in [9] is performed assuming unbounded integer variables.

However, SystemC, SpecC and Handel-C all offer an extensive set of bit-wise operators, which are not supported by this approach. Bit-wise operators perform bit-wise manipulations on the operands, i.e., a bit-wise operator combines a bit in one operand with its corresponding bit in the other operand to calculate one bit for the result. For example, the bit-wise `AND (&)` of the two bit-vectors `1101` and `0111` is `0101`. We use the term 'bit-wise operator" to refer to the various operations possible on one or more bit-vectors. These operations include *shift* operators, which shift the bits in a given operand to either the left or the right by a specified number of positions, the *concatenation* operator, which combines two or more bit-vectors, and the *extraction* operator, which extracts bits from a bit-vector from the specified positions. At the system-level, the use of these bit-level constructs is ubiquitous.

An algorithm that preserves the bit-vector semantics during predicate abstraction is presented in [13, 14]: A SAT solver is used to compute an abstraction of an ANSI-C program. The approach supports 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. A version of SLAM that implements SAT-based predicate abstraction is reported to have found a previously unknown bug in a Windows device driver in [18]. The bug depends on bit-vector operations, and thus, was not found using integer semantics.

Contribution. This paper presents a method to verify a concurrent SpecC system description with communication through shared variables using the CEGAR paradigm. This includes reasoning about the concurrency constructs (such as `par`, `wait`, and `notify`) and the

bit-vector operators found in SpecC. The `par` construct starts the concurrent execution of threads, the `wait` construct suspends the execution of a thread until a given event occurs, the `notify` construct generates the events specified as arguments. Before the verification starts, a pre-processing step is performed, which replaces all occurrences of the `par` construct with equivalent static threads. Each thread consists of only guarded `goto`, assignment, and few concurrency related statements. The verification is performed on the pre-processed program using the CEGAR loop.

We describe the four steps that form the CEGAR loop, namely abstraction, model checking, simulation, refinement. All steps of the CEGAR loop are completely automated, and do not require any manual intervention. The abstraction of the given program is computed using SAT-based predicate abstraction as introduced earlier in [13]. Each thread of control is abstracted separately. The abstractions preserve the bit-vector semantics of SpecC, and all SpecC bit-vector operators are supported.

The abstractions of the individual threads are then combined to obtain a concurrent finite state model. This model is verified using a conventional BDD-based symbolic model checker. If the model checker reports that the property holds on the abstract model, then the property holds on the given SpecC program. In this case, the CEGAR loop outputs "property holds" and terminates. Otherwise, an abstract counterexample is produced by the model checker.

The abstract counterexample corresponds to a sequence of statements *Seq* and includes the interleavings between the various threads in the pre-processed SpecC program. The purpose of the simulation step is to check if *Seq* corresponds to any *concrete counterexample* (i.e., a real bug) in the given program or not. The simulation is done with a BMC-like computation using a SAT solver. If *Seq* is a trace of a real bug, then the bug is reported and the CEGAR loop terminates. Otherwise, *Seq* denotes a sequence of statements that cannot occur during any execution of the given program. Such a sequence of statements is called a *spurious counterexample*.

The spurious counterexample is analyzed by the refinement procedure to produce additional information, which is used to make later abstractions more precise, that is, to remove the spurious counterexample. In the context of predicate abstraction, the refinement step is used to discover new predicates. We perform the refinement using a backwards *weakest precondition* [22] computation over *Seq*. This method guarantees that the counterexample *Seq* is not obtained again. The prior work, which is less general, targets either sequential programs, or disregards bit-vectors, or does not support communication through shared variables.

Optionally, a low level design (circuit level) can be added during the verification process. The low-level design may be used for two purposes:

1. The low-level design can be used to check refinement, i.e., 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. [47]. 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 [38]. 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 [33], Bounded Model Checking (BMC) [6, 7] is applied to both a circuit and an ANSI-C program. The approach is restricted to sequential ANSI-C programs; no support 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 [35], which is too large for many industrial designs. A Bounded Model Checker for concurrent ANSI-C programs with communication through shared variables is presented by Rabinovitz and Grumberg in [46].

In [32], the authors apply SAT-based predicate abstraction to the equivalence checking problem. The high-level language used is ANSI-C, not a system level language, and no concurrency is supported.

The concept of verifying the equivalence of a software implementation and a synchronous transition system was introduced by Pnueli, Siegel, and Shtrichman [43]. Since the target code is generated automatically by a compiler, the C program is assumed to have a specific form.

Clarke et al. [12] 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.

Several methods address the problem of scalability in the presence of threads and non-deterministically chosen data via forms of decomposition [2, 26]. Henzinger et al. apply an algorithm for model checking safety properties of concurrent software for automatic race detection in multi-threaded C programs [26]. However, the analysis of Henzinger et al. does not cover hardware-like bit-vector manipulation. These techniques usually either sacrifice some amount of completeness or require small amounts of intervention from the user. The advantage of these approaches is that the analysis is much more scalable. Unsound approaches have also proved successful in finding bugs in concurrent programs. For example, Qadeer & Rehof [45] note that many bugs can be found when the analysis is limited to execution traces with only a small number of context-switches between concurrent threads. This analysis supports recursive programs.

Qadeer et al. [44] 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.

Forms of partial-order reduction for explicit-state model checking (examples include [24, 28]) have been particularly effective for verifying programs and protocols with many threads. For example, Ball, Chaki and Rajamani [3] describe a partial-order reduction based explicit state model checker, called Beacon, for asynchronous Boolean programs. Beacon, however, is overly sensitive to the occurrence of symbolic data.

Outline. In Section 2, we provide a background on SpecC, and describe how we pre-process the SpecC program for verification. Section 3 formalizes the semantics of the synchronization

constructs found in SpecC. 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 [23] 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 that defines 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`, `interface`, and `channel` declarations. The syntax of a behavior is similar to the syntax of 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 semantics. 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 (as specified). The OR semantics means that the wait construct suspends the execution of the current thread until at least one of the events occurs. The AND semantics means that the wait construct suspends the execution of the current thread until all the given events have occurred. A particular ordering between events is not required.
- The `notify` construct generates the events specified as arguments. The generated events are delivered to all the threads that are currently waiting or sensitive to the notified events. This allows resuming the execution of threads that currently (or in the future) wait on the generated events.
- 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. Therefore, an event must not be used in any expression. Events can be used only with certain constructs such as `wait` and `notify`.

```
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;
   }
};
```

Fig. 1 Small SpecC program *P* from the SpecC language reference manual [23]

Example 1. The SpecC program of Fig. 1 shows the use of the `wait` and `notify` constructs described above. The 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 the 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 requires 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 all threads, which activates those threads that were waiting on any of them. This is why in the example above *B* is guaranteed to receive the event sent by *A*. Note that the `wait` can occur any time after the `notify`. In particular, the following sequence is guaranteed not to deadlock.[1]

```
notify e;
wait e;
```

Example 2. The SpecC program of Fig. 2(a) shows a channel class *C* that implements an interface *I*. The interface *I* specifies the *send* and *receive* methods. The channel class *C* provides a simple implementation of the *send* and *receive* methods by use of an integer variable *d*. Intuitively, the methods of a channel specify the communication protocol, whereas the variables of a channel resemble the communication media.

[1] We are grateful to Masahiro Fujita for clarifying this issue.

```
interface I {
   void send(int);
   int receive(void);
};

channel C implements I {
   int d;
   void send(int x) {
      d = x;
   };
   int receive(void) {
      return d;
   };
};
```

(a)

```
void C::send(int x) {
   lock(C);
   d = x;
   unlock(C);
}

int C::receive(void) {
   int temp;
   lock(C);
   temp = d;
   unlock(C);
   return temp;
}
```

(b)

Fig. 2 (a) Example of a channel *C* taken from [23]. (b) Making implicit locks in channel methods explicit by adding lock and unlock statements

2.2 Pre-processing

In this section we describe the steps used to simplify the given SpecC program before it is given to the verification tool. All transformations are performed automatically. 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. Figure 2(b) illustrates the *send* and *receive* methods implemented by channel *C* after making the locks explicit.

This is followed by the removal of side effects, that is, pre- and post-increment operators, the compound assignment operators (such as `x+ = y`) and the assignment operators that are used as expressions (such as `(x = (y = z))`), 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`, simple 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. 3(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. 3(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

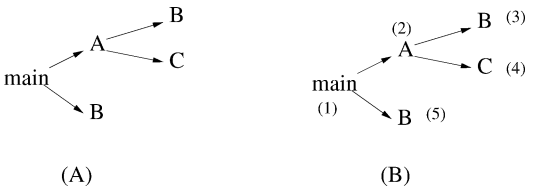

Fig. 3 (A) Nested par structure (B) DFS numbering starting from main

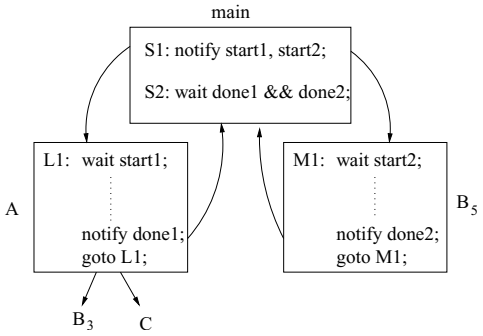

Fig. 4 Replacement of `par` in the `main` thread

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. 3(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. 3(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. 4. The par statement in the main thread is replaced by the following statements:

$$\texttt{notify } start_1,\ start_2;$$
$$\texttt{wait } done_1 \ \&\& \ done_2;$$

The statements `wait` $start_1$ and `wait` $start_2$ are added to the beginning 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.

The `goto` statements at the end of the threads A and B_5 in Fig. 4 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. 4 are assumed to be true. Recall that thread A starts the concurrent execution of the threads B_3 and C by using a `par` statement. This `par` statement is also replaced by the `wait` and `notify` statements as done for the `main` thread to start the threads B_3 and C. This is depicted by arrows from A to B_3 and C in Fig. 4.

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. This pre-processing greatly simplifies the construction of the formal models that can subsequently be given to standard program verification tools.

3 Formal semantics

The SpecC execution semantics have been described by Dömer et al. [23] using the time interval formalism. Mueller et al. [41] formalized the execution semantics of SpecC using distributed Abstract State Machines. In this section, we describe the operational semantics that we use. Given a SpecC program P, we define 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 if event e has occurred. This definition of E treats all events as global, that is, we assume that each event can potentially be used by any thread in `wait` or `notify`-like constructs. This makes the description of formal semantics easier. In practice, the number of event bits can be reduced if an event is used only by a subset of the threads. 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:

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

The transition relation $R(s, s')$ relates two states s and s', where s' is obtained by choosing one of the threads P_i non-deterministically 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 the 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'.

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

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'.

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

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.

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

If $\Gamma(s, i)$ is an assignment statement of the form $(\texttt{v} := \texttt{exp})$, 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.

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

If $\Gamma(s, i)$ is a wait statement of the form $(\texttt{wait}, \text{AND}, W)$, 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.[2] 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.

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

[2] Thus, we implement a form of busy-waiting—this is wasteful if execution is the goal, but simplifies the model if the goal is model checking.

The treatment 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.

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

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. Note that both $read(i, s)$ and $write(i, s)$ may depend on the valuations of pointer variables in case the statement that is to be executed contains pointer dereferencing operators.

The presence of a `RW` or `WW` conflict can be cast as the following safety property:

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

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 or less 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. In the following, we will describe a verification method that is based on this assumption as it abstracts programs statement by statement.

4 Counterexample guided abstraction refinement loop for SpecC

4.1 Predicate abstraction

We verify the SpecC program using counterexample guided abstraction refinement (CEGAR). We perform a predicate abstraction [25], 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 [17], 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 transition system $\hat{T}$, and we denote the transition relation of $\hat{T}$ by $\hat{R}$.

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

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 initial set of states $I(x)$ is abstracted as follows: an abstract state $\hat{x}$ is an initial state in the abstract model if there exists a concrete state x that is an initial state in the concrete model and is abstracted to $\hat{x}$.

$$\hat{I}(\hat{x}) := \exists x \in S : \alpha(x) = \hat{x} \wedge I(x)$$

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 $\hat{x}$.

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

Thus, if $\hat{P}$ holds on all reachable states of the abstract model, P 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 [21] to compute the abstraction. This approach suffers from the fact that errors caused by bit-vector overflow may remain undetected. As a motivating example, the formula $(x - y > 0) \iff (x > y)$ obviously holds if x and y are integers, but no longer holds once x and y are interpreted as bit-vectors, due to possible overflow on the subtraction operation.

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 [13], the authors propose to use a SAT solver to compute the abstraction of a sequential ANSI-C program. We implement this approach for computing abstractions of SpecC programs. The method 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 of the SpecC program. 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). Similarly, no abstraction is done for the `lock` and `unlock` statements. They are modeled in the abstract model by using new Boolean variables (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 $u \in V \setminus \{v\}$ 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.

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

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 [13]. The result is DNF over the predicates. There are a number of ways to improve this basic algorithm,

e.g., *predicate partitioning* as described in [30]. However, these techniques are beyond the scope of this article.

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 [10, 39].

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. The bit e_1 is used to transmit e to P_1, and the bit e_2 is used to transmit e to P_2. Let l_1 and l_2 denote the program counter (pc) values of the `wait` e statement in P_1 and of the `notify` e statement in P_2, respectively. The SMV statements generated for the `wait` e statement in P_1 are as follows:

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

TRANS pc1 = l1 ∧ e1 → !next(e1)        //resetting e1
```

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

$$\texttt{TRANS}\ pc_2 = l_2 \rightarrow \texttt{next}(e_1) \wedge \texttt{next}(e_2)$$

Let C denote a channel. As illustrated above, the implicit lock that guards the channel is translated into explicit `lock` and `unlock` statements. The translation of the `lock(C)`, `unlock(C)` statements into SMV is based on a new Boolean variable for each lock. Let b denote the Boolean variable corresponding to the lock for C. Let l_1 and l_2 denote the program counter values of the `lock(C)` and `unlock(C)` statements, respectively. The

SMV statements generated for the `lock(C)` statement are as follows:

```
ASSIGN next(pc) :=

  case pc = l1 :                 //lock statement
    case !b : l1 + 1;            //lock available, increment program counter
         b : l1;                 //lock held by some other thread
    esac;
  ...
  esac;

TRANS pc = l1 ∧ !b → next(b)     //acquire lock
```

The SMV statement generated for the `unlock(C)` statement is as follows:

$$\texttt{TRANS}\ pc = l_2 \rightarrow !\texttt{next}(b) \qquad //\text{release lock}$$

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. If the branch is not taken in the abstract trace, g is negated. Furthermore, a conjunct is added that constrains the values of the variables to be equal to the previous values:

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

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

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

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

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

The formal definition of Σ_i for `wait, lock` and `unlock` 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 satisfiability 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.

If s is a `goto` statement with guard g, we write `assume g` or `assume` $\neg$`g` to denote the branch of the `goto` statement that is executed in the abstract trace.

Example. Let S denote the following counterexample trace.

```
assume(y == 2);
x = y;
assume(!(x > 0));
...
```

It is checked if S is a real counterexample (bug) or a spurious counterexample by checking the satisfiability of the following formulas:

$$\Sigma_1 := (y_1 = 2) \wedge (x_1 = x_0) \wedge (y_1 = y_0)$$
$$\Sigma_2 := \Sigma_1 \wedge (x_2 = y_1) \wedge (y_2 = y_1)$$
$$\Sigma_3 := \Sigma_2 \wedge (x_3 = x_2) \wedge (y_3 = y_2) \wedge \neg(x_3 > 0)$$

We use a SAT solver to check the satisfiability of the above formulas. Observe that Σ_1, Σ_2 are satisfiable, but Σ_3 is unsatisfiable. Thus, trace S is a spurious counterexample and the guard that caused the last SAT instance to be unsatisfiable is $!(x > 0)$.

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* [22] of the guard g that caused the last SAT-instance Σ to be unsatisfiable; as above, g is negated if the branch is not taken.

Formally, let formula ϕ describe a set of program states, namely, the states in which the value of program variables satisfy ϕ. The weakest pre-condition of a formula ϕ with respect to a statement s is the weakest formula whose truth before the execution of s entails the

truth of ϕ after s terminates. We denote the weakest pre-condition of ϕ with respect to s by $WP(\phi, s)$. It is defined as follows:

- If s is an assignment statement of the form `v = exp;`, then the weakest pre-condition of ϕ with respect to s, is obtained from ϕ by replacing every occurrence of v in ϕ with exp.
- If s is a `goto` statement with guard g, we write `assume g` or `assume ¬g` to denote the branch of the `goto` statement that is executed. The weakest pre-condition of ϕ with respect to `assume g` is $\phi \wedge g$.
- If s is a `notify, wait, lock` or `unlock` statement, then the weakest pre-condition of ϕ with respect to s is ϕ itself.

The weakest pre-condition operator is extended to a sequence of statements by $WP(\phi, s_1; s_2) = WP\ (WP\ (\phi, s_2), s_1)$. 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 are obtained by extracting the atomic predicates (Boolean expressions) from the weakest pre-condition. For example, if the weakest pre-condition is $x > y \wedge x == 3$, we obtain the two predicates $x > y$ and $x == 3$. Both the computation of the weakest pre-condition and the extraction of the atomic predicates are completely automated. The new predicates are added to the previous set of predicates. This method guarantees that future abstract models do not contain the same spurious counterexample.

Example. Let S denote the following counterexample trace.

```
assume(y == 2);
x = y;
assume(!(x > 0));
...
```

As described in the previous example, S is a spurious counterexample, and the guard g that makes the last SAT instance unsatisfiable is $!(x > 0)$. To eliminate this spurious counterexample, the weakest pre-condition of g is computed with respect to the statements occurring before g in S. We denote these statements by $S' :=$ `assume(y == 2); x = y;`. The weakest pre-condition of g with respect to S' is:

$$WP(\phi, S') = WP(WP(!(x > 0), x = y), \texttt{assume}(y == 2))$$
$$WP(\phi, S') = WP(!(y > 0), \texttt{assume}(y == 2))$$
$$WP(\phi, S') = !(y > 0) \wedge y == 2$$

The weakest pre-condition of S' with respect to ϕ is $!(y > 0) \wedge y == 2$. There are two new atomic predicates that occur in the weakest pre-condition: $y > 0$ and $y == 2$.

The next example illustrates the operation of the CEGAR loop using a small SpecC program. Note that all steps of the CEGAR loop are automatic and do not require user intervention.

Example. The SpecC program given in Fig. 5(a) has two threads A and B, which communicate using events e_1, e_2. For simplicity we omit some syntax irrelevant to this example and the `Main` thread, which starts the threads A and B. The program operates as follows: thread A writes 42 into the shared variable x. After that, thread A generates an event e_1, which matches with the `wait e1` statement in thread B. Next, thread B performs some

```
event e1, e2;
int x;

behavior A () {
A1:  x = 42;
A2:  notify e1;
A3:  wait e2;
A4:  if !(x > 0)
A5:     ERROR:;
}

behavior B () {
     int y;
B1:  wait e1;
B2:  if (y == 2) {
B3:     x = y;
B4:  }
B5:  notify e2;
}
```

```
event e1, e2;
bool p1; /* x>0 */

process A () {
A1:  p1 = true;
A2:  notify e1;
A3:  wait e2;
A4:  if !(p1)
A5:     ERROR:;
}

process B () {

B1:  wait e1;
B2:  if (*) {
B3:     p1 = *;
B4:  }
B5:  notify e2;
}
```

```
event e1, e2;
bool p1; /* x>0 */
bool p2; /* y>0 */
bool p3; /* y==2 */

process A () {
A1:  p1 = true;
A2:  notify e1;
A3:  wait e2;
A4:  if !(p1)
A5:     ERROR:;
}

process B () {

B1:  wait e1;
B2:  if (*) {
B2:     p3 = true;
B2:     p2 = true;
B3:     p1 = p2;
B4:  }
B5:  notify e2;
}
```

Fig. 5 (a) A SpecC program with two threads. (b) Abstraction with respect to the predicate set $\{x > 0\}$ (c) Abstraction with respect to the predicate set $\{x > 0, y > 0, y == 2\}$

computation on x. Once thread B completes its computation, it generates an event e_2. Upon receiving the event e_2, thread A checks the value of the guard !(x > 0). If this guard is true, an ERROR label is reached.

We use the CEGAR loop to show that the ERROR label is not reachable in the given program. The first step of this loop is to create an abstraction of the concrete program using an initial set of predicates. In practice one starts with an empty set of predicates and new predicates are discovered through refinement. For this example, we suppose that the initial set of predicates contains the predicate $x > 0$. The abstraction with respect to the predicate $x > 0$ is shown in Fig. 5(b). Each thread of the SpecC program is abstracted separately to a *process* in the abstraction. The control flow of the concrete program is preserved in the abstraction. Since the event structure is not abstracted, wait and notify statements from the concrete program are directly copied in the abstraction. For simplicity we do not show the actual SMV code of the abstract model.

In the abstract model, the predicate $x > 0$ is represented by the Boolean variable p_1. Each statement in the abstraction represents the effect of the corresponding concrete statement on the set of predicates. For example, the statement x = 42 in thread A makes the predicate $x > 0$ true at the program location A1 in the abstraction. The statement x = y in thread B assigns a non-deterministically chosen Boolean value (*) to $x > 0$ at program location B3. This is because the current set of predicates contains no information about y.

Running a model checker on the abstraction in Fig. 5(b) returns an abstract counterexample shown in Fig. 6(a). An assume statement is used to specify which branch of the if statement is taken in the counterexample. In this counterexample, p_1 is assigned false

```
A1: p1 = true;
A2: notify e1;
B1: wait e1;
B2: assume (true);
B3: p1 = false;
B5: notify e2;
A3: wait e2;
A4: assume (!p1);
A5: ERROR;
```

```
A1: x = 42;
A2: notify e1;
B1: wait e1;
B2: assume (y == 2);
B3: x = y;
B5: notify e2;
A3: wait e2;
A4: assume !(x > 0);
```

Fig. 6 (a) Counterexample obtained after model checking the abstraction in Fig. 5(b). (b) Corresponding counterexample in the concrete program Fig. 5(a)

at location B3, making the condition $!p_1$ in A4 true. This makes the ERROR label reachable in the abstract model. The sequence of SpecC statements corresponding to this abstract counterexample is shown in Fig. 6(b). Observe that this sequence of statements is not feasible in any execution of the concrete program. This is because at location B2, we assume y to be equal to 2. At location B3, the variable x is assigned to the value of y, and thus, the guard $!(x > 0)$ in location A4 must evaluate to false. Thus, the abstract counterexample is spurious. Since the guard $!(x > 0)$ in location A4 causes the simulation to fail, we compute its weakest pre-condition with respect to the statements shown in Fig. 6(b). The weakest precondition $!(y > 0) \wedge (y == 2)$ results in two new atomic predicates $y > 0$, $y == 2$.

The abstraction using the new predicate set $\{x > 0, y > 0, y == 2\}$ is shown in Fig. 5(c). Observe that inside the true branch (assume y == 2) of the if statement at location B2, the predicates $y == 2$ (p_3) and $y > 0$ (p_2) are assigned true. The effect of the statement x = y in thread B is to assign the value of predicate p_1 the value of p_2 in the abstraction. Model checking of this abstraction is successful, that is, the ERROR label is shown not to be reachable in the abstraction. Thus, the ERROR label is not reachable in the concrete program.

5 Experimental results

We have implemented the algorithm described above in a tool called SATABS. The implementation includes front-ends for ANSI-C, SpecC, Verilog, and SystemC. We make our implementation available for experimentation by other researchers.[3]

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 Tables 1 and 2. The PIPE benchmarks are 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 is an assertion that the data that was put in the pipeline matches the data that comes out of the pipeline. The run-time 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,

[3] http://www.inf.ethz.ch/personal/daniekro/satabs/

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

Benchmark	Threads	Bug length	Predicates	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

Table 2 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

Benchmark	Threads	Bug length	Predicates	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

we experimented with two different implementations, CMU SMV [39] and NuSMV [10]. NuSMV clearly outperforms CMU SMV, and therefore we only report the NuSMV time. Both model checkers show exponential run-time 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. In contrast to the benchmarks above, the `ALUPIPE` benchmarks require predicates that contain complex arithmetic. In case of the passing properties (`ALUPIPE A` and `ALUPIPE C`), the run-time is dominated by the abstraction computation phase, and in the case of the failing property (`ALUPIPE B`), the run-time is dominated by the simulation phase.

Experiments with SPIN. We also experimented with SPIN [27], which is an explicit state model checker with partial order reduction. On a large number of threads and a small number of predicates, SPIN clearly outperforms NuSMV due to its partial order reduction algorithm. However, SPIN quickly runs out of memory as soon as even a moderate (>30) predicates are

used, as all states are represented explicitly. As future work, we plan to investigate symbolic model checkers that implement partial order reduction.

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. SpecC offers an extensive set of bitwise operators. Our algorithm models the bit-vector semantics of the language accurately by means of a direct, bit-wise encoding of the variables and operators into propositional logic. It 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. We implemented a model checker that integrates partial order reduction into a SAT-solver to verify the abstract models [19].

Predicate discovery for abstraction-refinement is still an open area of research. We described how new predicates can be discovered using weakest pre-conditions. This is theoretically sufficient for making sure that CEGAR loop makes progress. However, it may not be practical for certain examples where a large (intractable) number of refinement iterations are needed to get the "right" set of predicates. An alternative technique for discovering new predicates is based on interpolation [31].

Among the system level languages, SystemC stands out since it is the only language based on C++ and not on plain ANSI-C. We have implemented the predicate abstraction refinement loop for models extracted from SystemC [34]. In contrast to SpecC, SystemC does not permit arbitrary interleavings, but only upon thread termination or the execution of a `wait` statement. This reduces the complexity of the model checking problem, but does not support modeling of real-time operating systems. The paper also describes how to reduce the complexity of verification by statically scheduling (merging) particular threads. As future work, we plan to investigate opportunities for exploiting the object structure of SystemC models.

Acknowledgments We thank Masahiro Fujita for numerous clarifications of the semantics of SpecC, and the anonymous referees for helpful suggestions.

References

1. SystemC, http://www.systemc.org
2. Alur R, Henzinger TA, Mang F, Qadeer S, Rajamani SK, Tasiran S (1998) MOCHA: Modularity in model checking. In: Proceedings of the 10th international conference on computer-Aided verification (CAV), vol 1427 of Lecture notes in computer science, pp 521–525
3. Ball T, Chaki S, Rajamani SK (2001) Parameterized verification of multithreaded software libraries. In: Proceedings of the 7th international conference on tools and algorithms for the construction and analysis of systems (TACAS), vol 2031 of lecture notes in computer science, pp 158–173
4. Ball T, Rajamani SK (2000) Boolean programs: A model and process for software analysis. Technical Report 2000-14, Microsoft Research
5. Ball, T, Rajamani SK (2001) Automatically validating temporal safety properties of interfaces. In: The 8th International SPIN workshop on model checking of software, vol 2057 of lecture notes in computer science, pp 103–122

6. Biere A, Cimatti A, Clarke EM, Fujita M, Yhu Y (1999a) Symbolic model checking using SAT procedures instead of BDDs. In: Proceedings of the 36th conference on design automation conference (DAC), pp 317–320
7. Biere A, Cimatti A, Clarke EM, Yhu Y (1999b) Symbolic model checking without BDDs. In: Proceedings of the 5th international conference on tools and algorithms for construction and analysis of systems (TACAS), vol 1579 of lecture notes in computer science, pp 193–207
8. Burch JR, Clarke EM, McMillan KL, Dill DL, Hwang LJ (1992) Symbolic model checking: 10^{20} states and beyond. Inf Comput 98(2):142–170
9. Chaki S, Clarke E, Groce A, Ouaknine J, Strichman O, Yorav K (2004) Efficient verification of sequential and concurrent C programs. Form Meth Syst Des (FMSD) 25(2–3):129–166
10. Cimatti A, Clarke E, Giunchiglia E, Giunchiglia F, Pistore M, Roveri M, Sebastiani R, Tacchella A (2002) NuSMV Version 2: An opensource tool for symbolic model checking. In: Proceedings of the 14th international conference on computer aided verification (CAV), vol 2404 of lecture notes in computer science, pp 359–364
11. Clarke E, Grumberg O, Peled D (1999) Model checking. MIT Press
12. Clarke E, Grumberg O, Talupur M, Wang D (2003) High level verification of control intensive systems using predicate abstraction. In: Proceedings of the 1st ACM and IEEE international conference on formal methods and models for co-design (MEMOCODE), pp 55–64
13. Clarke E, Kroening D, Sharygina N, Yorav K (2004) Predicate abstraction of ANSI-C programs using SAT. Formal Methods Syst Des (FMSD) 25:105–127
14. Clarke E, Kroening D, Sharygina N, Yorav K (2005) SATABS: SAT-based predicate abstraction for ANSI-C. In: Proceedings of the 11th international conference on tools and algorithms for the construction and analysis of systems (TACAS), vol 3440 of lecture notes in computer science, pp 570–574
15. Clarke EM, Emerson EA (1981) Synthesis of synchronization skeletons for branching time temporal logic. In: Logic of programs: Workshop, vol 131 of Lecture notes in computer science, pp 52–71
16. Clarke EM, Grumberg O, Jha S, Lu Y, Veith H (2000) Counterexample-guided abstraction refinement. In: Proceedings of the 12th international conference on computer aided verification (CAV), vol 1855 of lecture notes in computer science, pp 154–169
17. Clarke EM, Grumberg O, Long DE (1992) Model checking and abstraction. In: Proceedings of the 19th symposium on principles of programming languages (POPL), pp 342–354
18. Cook B, Kroening D, Sharygina N (2005a) Cogent: Accurate theorem proving for program verification. In: Etessami K, Rajamani SK (eds) Proceedings of the 19th international conference on computer aided verification (CAV), vol. 3576 of lecture notes in computer science, pp 296–300
19. Cook B, Kroening D, Sharygina N (2005b) Symbolic model checking for asynchronous boolean programs. In: Godefroid P (ed) Proceedings of the 12th international SPIN workshop, vol 3639 of lecture notes in computer science, pp 75–90
20. Copty F, Fix L, Fraer R, Giunchiglia E, Kamhi G, Tacchella A, Vardi MY (2001) Benefits of bounded model checking at an industrial setting. In: Berry G, Comon H, Finkel A (eds) Proceedings of the 13th international conference on computer aided verification (CAV), pp 436–453
21. Detlefs D, Nelson G, Saxe JB (2003) Simplify: A theorem prover for program checking. Technical Report HPL-2003-148, HP Labs
22. Dijkstra E (1976) A discipline of programming. Prentice Hall
23. Dömer R, Gerstlauer A, Gajski D (2002) SpecC language reference manual, Version 2.0. http://www.specc.org/
24. Flanagan C, Godefroid P (2005) Dynamic partial-order reduction for model checking software. In: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on principles of programming languages (POPL), pp 110–121
25. Graf S, Saïdi H (1997) Construction of abstract state graphs with PVS. In: Grumberg O (ed) Proceedings of the 9th international conference on computer aided verification (CAV), vol 1254 of lecture notes in computer science, pp 72–83
26. Henzinger TA, Jhala R, Majumdar R, Qadeer S (2003) Thread modular abstraction refinement. In: Proceedings of the 15th international conference on computer-aided verification (CAV), vol 2725 of lecture notes in computer science, pp 262–274
27. Holzmann GJ (1997) The model checker SPIN. IEEE Trans Softw Eng 23(5):279–295
28. Holzmann GJ, Peled D (1994) An improvement in formal verification. In: Proceedings of the 7th IFIP WG6.1 international conference on formal description techniques, pp 197–211
29. Jain H, Clarke E, Kroening D (2004) Verification of SpecC and verilog using predicate abstraction. In: Proceedings of the 2nd ACM and IEEE international conference on formal methods and models for co-design (MEMOCODE), pp 7–16

30. Jain H, Kroening D, Sharygina N, Clarke E (2005) Word level predicate abstraction and refinement for verifying RTL Verilog. In: Proceedings of the 42nd design automation conference (DAC), pp 445–450
31. Jhala R, McMillan KL (2006) A practical and complete approach to predicate refinement. In: TACAS, vol. 3920 of lecture notes in computer science, pp 459–473
32. Kroening, D, Clarke E (2004) Checking consistency of C and Verilog using predicate abstraction and induction. In: Proceedings of the 2004 IEEE/ACM international conference on computer-aided design (ICCAD), pp 66–72
33. Kroening D, Clarke E, Yorav K (2003) Behavioral consistency of C and VERILOG programs using bounded model checking. In: Proceedings of the 40th design automation conference (DAC), pp 368–371
34. Kroening D, Sharygina N (2005) Formal verification of system C by automatic hardware/software partitioning. In: Proceedings of the 3rd ACM and IEEE international conference on formal methods and models for co-design (MEMOCODE), pp 101–110
35. Kroening D, Strichman O (2003) Efficient computation of recurrence diameters. In: Zuck L, Attie P, Cortesi A, Mukhopadhyay S (eds) Proceedings of the 4th international conference on verification, model checking, and abstract interpretation (VMCAI), vol 2575 of lecture notes in computer science, pp 298–309
36. Ku, D, DeMicheli G (1990) HardwareC—A language for hardware design (Version 2.0). Technical Report CSL-TR-90-419, Stanford University
37. Kurshan RP (1994) Computer-aided verification of coordinating processes: The automata-theoretic approach. Princeton University Press
38. Matsumoto T, Saito H, Fujita M (2003) Equivalence checking of C based hardware descriptions by using symbolic simulation and program slicer. In: International workshop on logic and synthesis (IWLS'03)
39. McMillan K: CMU Symbolic Model Verifier, http://www.cs.cmu.edu/∼modelcheck/smv.html
40. Moskewicz MW, Madigan CF, Zhao Y, Zhang L, Malik S (2001) Chaff: Engineering an efficient SAT solver. In: Proceedings of the 38th design automation conference (DAC), pp 530–535
41. Mueller W, Dömer R, Gerstlauer A (2002) The formal execution semantics of specC. In: Proceedings of the 15th international symposium on system synthesis, pp 150–155
42. Page I (1996) Constructing hardware-software systems from a single description. J VLSI Signal Process 12(1):87–107
43. Pnueli A, Shtrichman O, Siegel M (1998) The code validation tool CVT: Automatic verification of a compilation process. Int J Softw Tools Technol Transf (STTT) 2(2):192–201
44. Qadeer S, Rajamani SK, Rehof J (2004) Summarizing procedures in concurrent programs. In: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on principles of programming languages (POPL), pp 245–255
45. Qadeer, S, Rehof J (2005) Context-bounded model checking of concurrent software. In: Proceedings of the 11th international conference on tools and algorithms for the construction and analysis of systems (TACAS), vol 3440 of lecture notes in computer science, pp 3–107
46. Rabinovitz I, Grumberg O (2005) Bounded model checking of concurrent programs. In: Proceedings of the 17th international conference on computer aided verification (CAV), vol 3576 of lecture notes in computer science, pp 82–97
47. Séméria L, Seawright A, Mehra R, Ng D, Ekanayake A, Pangrle B (2002) RTL C-based methodology for designing and verifying a multi-threaded processor. In: Proceedings of the 39th design automation conference (DAC), pp 123–128
48. Shtrichman O (2000) Tuning SAT checkers for bounded model checking. In: Emerson E, Sistla A (eds) Proceedings of the 12th international conference on computer aided verification (CAV), pp 480–494