Formal Verification of SystemC by Automatic Hardware/Software Partitioning

Daniel Kroening
Computer Systems Institute
ETH Zürich

Natasha Sharygina
Carnegie Mellon University
Software Engineering Institute

Abstract

Variants of general-purpose programming languages, like SystemC, are increasingly used to specify system designs that have both hardware and software parts. The system-level languages allow a flexible partitioning in the design of the hardware and software. Moreover, many properties depend on the combination of hardware and software and cannot be verified on either part alone. Existing tools either apply non-formal approaches or handle only the low-level parts of the language.

This papers presents a new technique that handles both hardware and software parts of a system description. This is done by automatically partitioning the uniform system description into synchronous (hardware) and asynchronous (software) parts. This technique has been implemented and applied to system level descriptions of several industrial examples. The hardware/software partitioning improves the performance of the verification compared to the monolithic approach.

1 Introduction

System designs have both hardware and software parts. Traditionally, the software component of a system design is written in a programming language like C or C++, while the hardware part is written in a hardware description language such as Verilog or VHDL.

This approach has several disadvantages. First of all, the designer is forced to learn and understand several languages. Second, at the beginning of the design process, it is often unclear which parts of the functionality are to be implemented in hardware or in software. If the partitioning of the design into hardware and software is to be changed later on, expensive and time consuming re-design becomes necessary. Furthermore, two different design languages usually break the verification tool flow. Many properties of the design only hold on the combination of particular software and hardware parts, and cannot be verified on either part alone.

This motivates the idea of using uniform system-level design languages. These languages offer various levels of abstraction, down from netlists up to highly abstract descriptions which hide low-level implementation details. 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 components of a system using a language that is well-known to engineers.

Several different projects have undertaken the task of extending the C language to support hardware specification. The earliest C-like hardware description language is HardwareC [29] from Stanford University, which 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 [34], developed at Oxford University, is very similar to SpecC, including the syntax for most of 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 of their features. All of them start with a high level of abstraction and bridge the gap to the lower levels by adding constructs like bit-vectors.

SystemC In contrast to these languages, the SystemC [37] language has a different approach. Historically, the SystemC language was used for low-level modeling of circuits only. For this low level, it has unique features such as four state logic signals (0, 1, Z, X) and multiple drivers for a single signal. The main motivation is efficient circuit simulation. The netlist of a SystemC model is obtained by synthesis in a similar manner as from synthesizable Verilog. SystemC also offers features not found in HDLs, such as fixed-point arithmetic. Hardware in SystemC is typically designed for synchronous execution. All components perform an action (event) at each execution step, synchronized by means of global clock signals.

The SystemC 2.0 standard aims at levels of abstraction above the RTL level, including parts that are implemented as concurrent software. Concurrent software systems typically use an asynchronous model of execution. The concurrent actions of the threads are executed in an arbitrary ordering, unless a particular ordering is enforced by the user. Thus, SystemC 2.0 adds constructs for the synchronization between concurrent threads. These constructs are based on *events*.

Formal Verification of System-Level Designs Concurrent software is notoriously error-prone. Approaches based on testing are usually fail to find important concurrency bugs. *Model Checking* [10, 15] is a formal verification technique which is now-days commonly used in the hardware design industry. It has been shown to be especially useful in verifying concurrency.

In order to support a hardware/software co-design, it is common to use interface constructs. In essence, an interface is modeled as a process that reads and writes communication signals from the hardware part. The coordination of an interface process with software is handled by the software communication mechanism, for example, message exchange via buffers. When hardware verification methods (based on synchronous execution models as in SMV, COSPAN, etc.) are used, the interface process not only uses the communication signals as synchronization events between hardware and software parts, but also performs a function of a scheduler. The scheduler is designed to model software asynchrony by self-looping, using nondeterminism as proposed in [30].

Traditionally, formal verification of hardware and software is done using different techniques, i.e., tools based on different algorithms, representations and principles. The tight coupling of hardware and software in a system model precludes verification of the hardware and software separately. Another problem is that for a single step in a software execution, there could be hundreds of steps done by the hardware. This difference makes traditional co-simulation inefficient. Thus, an efficient formal verification technique for designs that have both a hardware and a software part is highly desirable.

Related Work In [19], Drechsler and Große describe how to convert a gate-level model given in SystemC into BDDs. The BDD is used for forward reachability analysis. In [31], a very similar method is used to check equivalence between SystemC designs. Again, the design is synthesized using the SystemC compiler into a netlist represented using BDDs. Neither project supports any of the higher abstraction levels for description of system designs.

In [27], 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 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 [28], which is too large for many industrial designs.

Sakunkonchak and Fujita verify the synchronization mechanisms of a SpecC program in [36]. The data is encoded using difference decision diagrams. No hardware-level constructs are supported.

In [26], 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.

This paper builds on work described in [23], where SAT-based predicate abstraction is applied to a SpecC design. Abstraction is a principal method to combat the state space explosion problem. 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 relevant behaviors of the system.

Predicate abstraction [20, 16] is one of the most prominent techniques used in software verification. It is implemented by all major software model checking tools [6, 21, 3]. It abstracts data by only keeping track of certain predicates. Each predicate is represented by a Boolean variable in the abstract model, while the original variables are eliminated. The abstract program is created using *Existential Abstraction* [9], which is a conservative abstraction for reachability properties. If the property holds on the abstract model, it also holds on the original program.

The drawback of a conservative abstraction is that when model checking of the abstract program fails, it may produce a counterexample that does not correspond to a concrete counterexample. This is called a *spurious counterexample*. When a spurious counterexample is encountered, *refinement* is performed by adjusting the set of predicates in a way that eliminates this counterexample. This is automated by *Counterexample Guided Abstraction Refinement* [30, 3, 8].

Predicate abstraction tools used to employ theorem provers that implement a logic based on natural numbers in order to compute the abstraction. System-level languages like SystemC, however, require an accurate modeling of the program variables as bit-vectors. A SAT-based method that supports bit-vector semantics has been proposed in [13]. There are now versions of both SLAM [17] and COMFORT [22], the successor to MAGIC [6], with support for SAT as a decision procedure.

In [23], the whole SpecC program is treated as a single software program, no attempt is made to optimize the verification of the hardware part.

This paper uses the state/event-based notation introduced in [7] for modeling SystemC programs. The modeling framework consists of *labeled Kripke structures* (LKS), which are directed graphs in which states are labeled with atomic propositions and transitions are labeled with actions. The combined state-based and event-based notation has been explored by a number of researchers. De Nicola and Vaandrager [33], for instance, introduce 'doubly labeled transition systems', which are very similar to our LKSs. Kindler and Vesper [25] use a state/event-based temporal logic for Petri nets.

Contribution We formalize the semantics of SystemC by means of labeled Kripke structures. We use a model that allows labeling both states (with propositions) and the transitions (with actions for synchronization). In contrast to prior work, we automatically partition the design into a hardware and a software part. The resulting design performs fewer transitions and can be analyzed more efficiently by means of an automated and thread-modular abstraction refinement loop.

Outline We provide background information on SystemC, the computational model we use, and abstraction in section 2. We formalize the semantics of the subset of SystemC we handle in section 3. Our partitioning heuristic is described in section 4. Section 5 provides details on how to abstract the partitioned model and how to verify the abstract model. Experimental results are reported in section 6.

2 Background

2.1 SystemC

SystemC was originally developed for specification of low-level designs. The main motivation of SystemC is that a circuit model can be compiled using a regular C++ compiler, and then simulated efficiently. Synthesis tools can generate net-lists from SystemC. The SystemC language includes all constructs allowed in C++, albeit a very small subset is actually synthesizable.

Similar to the approach used by SpecC, modularization in SystemC is implemented by means of C++ classes: the SystemC construct `SC_MODULE` is simply a pre-processor macro for a class definition. Communication between the modules is done by means of pointers to shared variables. These pointers are set within the constructors and expected not to change during the runtime.

SystemC allows all C++ data types. In addition to that, types for arbitrary width bit-vectors and infinite precision fixed-point values are defined.

The behavior of a module is specified by defining one or more *Processes*. A process is either a *Method*, a *Thread*, or a *Clocked Thread*. Both methods and clocked threads are special cases of threads. We therefore focus the presentation on threads only (the SystemC standard makes this distinction for greater simulation and synthesis efficiency).

Syntactically, a process is a method of a module class. Similar to VHDL or Verilog, a process has a list of events that activate the process. This list of events is called the *sensitivity list* of the process. As soon as the event occurs, the process is activated and executes until the process terminates or suspends its execution by means of the `wait()` statement. The SystemC methods are special cases of processes that do not call `wait()`.

Events may either be generated explicitly by a thread (using the `notify()` statement or method), or implicitly by changing signal values. As an example, the positive edge of a clock signal is typically used as an activation event for processes that model clocked circuits.

These concepts are illustrated by an example in Figure 1: The listing shows a module `m` with two input ports of type `bool` and two threads. The first thread is sensitive to a change in the input value, and thus, could be used to model combinational circuitry. The second thread is sensitive to a positive clock edge, and thus, could be used for modeling a clocked circuit.

Semantics of Thread Execution The SystemC specification distinguishes three states of a thread: *run-*

```
SC_MODULE(m) {
  sc_in<bool> data_in;   // input port
  sc_in<bool> clock;     // input port
  sc_out<bool> data_out; // output port

  void thread1();
  void thread2();

  int i;

  SC_CTOR(m) { // Constructor
    SC_THREAD(thread1); // Thread Process
    // make thread sensitive to change of input
    sensitive << data_in;

    SC_THREAD(thread2); // Thread Process
    // make thread sensitive to clock
    sensitive_pos << clock;
  }
};
```

Figure 1. SystemC example: A module `m` (syntactic sugar for a class) with input/output ports and two threads.

```
SC_MODULE(M) {
  sc_event e;
  int data;

  SC_CTOR(M) {
    SC_THREAD(a);
    SC_THREAD(b);
  }

  void a() {
    data=1;
    cout << "setting data to " << data << endl;
    e.notify();
  }

  void b() {
    wait(e);
    cout << "data=" << data << endl;
  }
};
```

Figure 2. Process synchronization using an event `e`. This example may deadlock.

ning, *waiting*, and *runnable*. A running thread may generate more events. Similar to SpecC, events that are generated are recorded. This is done by changing the state of any thread sensitive to the event to *runnable*. A single event may trigger the execution of multiple threads.

A running thread may become a waiting thread by executing the `wait` statement. The scheduler chooses a thread among the runnable threads to resume execution. As in Verilog, the ordering in which the runnable threads are activated is chosen non-deterministically. It is important to note that no interleavings are done between the threads unless a `wait()` statement is executed. This is a major difference between SystemC and other system-level modeling languages such as SpecC, which allows arbitrary interleavings between the threads [1]. In particular, there is no need to consider data races as it done for SpecC in [23]. It is important to note that the synchronization does not happen upon the generation of the event, but only upon calling `wait`.

Delayed Notification An event is delivered as soon as it occurs. If the thread that is meant to receive it is already in the *runnable* state, the event is ignored. This can result in a deadlock, as illustrated in Figure 2. Initially, both threads are *runnable*. The scheduler may choose to execute either thread `a` or thread `b` first. If thread `a` is chosen first, it will generate the event `e`. However, thread `b` is not yet waiting for the event, and thus, the event is lost. The scheduler will execute thread `b` next, which will deadlock.

As an easy solution for such situations, the delivery of an event can be *delayed*. The example is changed as follows:

```
e.nodify_delayed();
```

In this case, the event is not delivered until all threads are in the *waiting* state, which fixes the deadlock. This can be implemented by adding a flag bit for each event, similar as described in [23].

2.2 Computational Model

A labeled Kripke structure [7] (LKS for short) is a 7-tuple $(S, \mathit{Init}, P, \mathcal{L}, T, \Sigma, \mathcal{E})$ with S a finite set of *states*, $\mathit{Init} \subseteq S$ a set of initial states, P a finite set of *atomic state propositions*, $\mathcal{L} : S \to 2^P$ a *state-labeling function*, $T \subseteq S \times S$ a transition relation, Σ a finite set (*alphabet*) of *events* (or *actions*), and $\mathcal{E} : T \to (2^\Sigma \setminus \{\emptyset\})$ a *transition-labeling function*. We often write $s \xrightarrow{A} s'$ to mean that $(s, s') \in T$ and $A \subseteq \mathcal{E}(s, s')$.[1] In case A is a singleton set $\{a\}$ we write $s \xrightarrow{a} s'$ rather than $s \xrightarrow{\{a\}} s'$. Note that both states and transitions are 'labeled', the former with sets of atomic propositions, and the latter with non-empty sets of actions.

A *path* $\pi = \langle s_1, a_1, s_2, a_2, \ldots \rangle$ of an LKS is an alternating infinite sequence of states and actions subject

[1]In keeping with standard mathematical practice, we write $\mathcal{E}(s, s')$ rather than the more cumbersome $\mathcal{E}((s, s'))$.

to the following: for each $i \geqslant 1$, $s_i \in S$, $a_i \in \Sigma$, and $s_i \xrightarrow{a_i} s_{i+1}$.

The *language* of an LKS M, denoted $L(M)$, consists of the set of maximal paths of M whose first state lies in the set *Init* of initial states of M.

2.3 Abstraction

Let $M = (S, \mathit{Init}, P, \mathcal{L}, T, \Sigma, \mathcal{E})$ and $\hat{M} = (S_{\hat{M}}, \mathit{Init}_{\hat{M}}, P_{\hat{M}}, \mathcal{L}_{\hat{M}}, T_{\hat{M}}, \Sigma_{\hat{M}}, \mathcal{E}_{\hat{M}})$ be two LKSs. We say that $\hat{M}$ is an *abstraction* of M, written $M \sqsubseteq \hat{M}$, iff

1. $P_{\hat{M}} \subseteq P$,
2. $\Sigma_{\hat{M}} = \Sigma$, and
3. For every path $\pi = \langle s_1, a_1, \ldots \rangle \in L(M)$ there exists a path $\pi' = \langle s'_1, a'_1, \ldots \rangle \in L(\hat{M})$ such that, for each $i \geqslant 1$, $a'_i = a_i$ and $\mathcal{L}_{\hat{M}}(s'_i) = \mathcal{L}(s_i) \cap P_{\hat{M}}$.

In other words, $\hat{M}$ is an abstraction of M if the 'propositional' language accepted by $\hat{M}$ contains the 'propositional' language of M, when restricted to the atomic propositions of $\hat{M}$. This is similar to the well-known notion of 'existential abstraction' for Kripke structures in which certain variables are hidden [8].

Two-way abstraction defines an equivalence relation $\sim$ on LKSs: $M \sim M'$ iff $M \sqsubseteq M'$ and $M' \sqsubseteq M$. We shall only be interested in LKSs up to $\sim$-equivalence.

2.4 Parallel Composition

We modify the notion of parallel composition in [7] to allow communication through shared variables.

Let $M_1 = (S_1, \mathit{Init}_1, P_1, \mathcal{L}_1, T_1, \Sigma_1, \mathcal{E}_1)$ and $M_2 = (S_2, \mathit{Init}_2, P_2, \mathcal{L}_2, T_2, \Sigma_2, \mathcal{E}_2)$ be two LKSs. We assume M_1 and M_2 share the same state space, i.e., $S = S_1 = S_2$, $P = P_1 = P_2$, and $\mathcal{L} = \mathcal{L}_1 = \mathcal{L}_2$. We denote by $s \xrightarrow{A}_i s'$ the fact that M_i can make a transition from s to s'.

The parallel composition of M_1 and M_2 is given by $M_1 \parallel M_2 = (S, \mathit{Init}_1 \cap \mathit{Init}_2, P, \mathcal{L}, T, \Sigma_1 \cup \Sigma_2, \mathcal{E})$, where T and $\mathcal{E}$ are such that $s \xrightarrow{A} s'$ iff $A \neq \emptyset$ and one of the following holds:

1. $A \subseteq \Sigma_1 \setminus \Sigma_2$ and $s \xrightarrow{A}_1 s'$,
2. $A \subseteq \Sigma_2 \setminus \Sigma_1$ and $s \xrightarrow{A}_2 s'$, or
3. $A \subseteq \Sigma_1 \cap \Sigma_2$ and $s \xrightarrow{A}_1 s'$ and $s \xrightarrow{A}_2 s'$.

In other words, components must synchronize on shared actions and proceed independently on local actions. This notion of parallel composition is similar to the definition used for CSP; see also [2].

3 Formal Semantics

3.1 Preparation

Similar as in [23], we pre-process the SystemC program. As most synthesis tools, we ignore the `sc_main` function and assume that there is a designated top-level module. We assume the set of threads can be determined statically, i.e., the module constructors must not contain non-trivial control flow, and the object (i.e., module) instances can be determined by finite unwinding. This is similar to what most synthesis tools expect [38]. Objects which are not SystemC modules may be constructed dynamically using `new`, however, and thus, we support designs with a true software part that is not synthesizable.

We support C++ templates by flattening: the template code is replicated, and the template parameters are substituted by the arguments given by the instance. Object construction and destruction is replaced by corresponding calls to the construction and destruction methods, respectively. Side effects are removed by syntactic transformations, the control flow statements (`if`, `while` and so on) are transformed into guarded `goto` statements. We currently do not support explicit exceptions using the `throw` statement. We support multiple inheritance and virtual member functions by adding function pointers to the compound type.

3.2 Transformation into an LKS

We formalize the semantics of SystemC using an LKS M_i for each thread. Let n denote the number of threads. The behavior of the whole SystemC program is given by the parallel composition $M_1 || \ldots || M_n$.

In SystemC, the only point of synchronization of threads and interleaving between the threads is the `wait` statement. Thus, we assume n global actions $\Omega := \{\omega_1, \ldots, \omega_n\} \subseteq \Sigma_i$ that are shared among all LKSs, and n local actions $\tau_i \in \Sigma_i$ with $\forall i \neq j. \tau_i \notin \Sigma_i$, i.e., τ_i is unique to M_i. If the thread is clear from the context, we simply write $s \xrightarrow{\tau} s'$ for a local transition of the tread.

Notation The global state space $S = S_1 = \ldots = S_n$ is spanned by the variables of all modules, a program counter PC_i for each thread, and the status $\sigma_i \in \{\mathit{waiting}, \mathit{runnable}, \mathit{running}\}$ of each thread. Thus, a state $s \in S$ is a three-tuple $(\overline{V}, \overline{PC}, \overline{\sigma})$ consisting of a vector $\overline{V}$ for the program variables, a vector $\overline{PC}$ for the PCs, and a vector $\overline{\sigma}$ for the status. Given a state $s \in S$, we denote the projection of the value of PC_i or σ_i from s as $s.PC_i$ and $s.\sigma_i$, respectively.

The execution of a statement by thread i increases the PC of thread i, while the other PCs remain unchanged. Let $\nu_i(\overline{\sigma})$ be a shorthand for $\overline{PC}'$ with $PC'_i = PC_{i+1}$ and $PC'_j = PC_j$ for $j \neq i$.

Initialization The SystemC 2.0 specification changed the initialization semantics. In the new version, all threads are initialized (i.e., started) once, but with no particular ordering. Thus, we define the set of initial states *Init* as the set of states $s \in S$ such that the PCs are set to the start of each thread and that one thread is running while the remaining threads are runnable.

$$s.PC_i = 0 \quad \wedge \quad s.\sigma_i \neq \mathit{waiting} \quad \wedge \\ (s.\sigma_i = \mathit{running} \longrightarrow \forall j \neq i.\, s.\sigma_j \neq \mathit{running})$$

Transition Relation The transition relation of LKS M_i is defined by splitting on the thread status. Let s denote the state from which the transition is made. If i is the running thread, i.e., $s.\sigma_i = \mathit{running}$, assume $\mathcal{P}_i(PC)$ denotes the instruction pointed to by PC in thread i. We do a case-split on the instruction, and thus, let I be a shorthand for $\mathcal{P}_i(s.PC_i)$.

- If I is a `notify` statement, the thread i makes a τ-transition and changes the global state accordingly. Note that no synchronization with other threads is performed. The set of threads that is activated by events caused by I is denoted by $\mathcal{A}(I)$. Formally,

  $$I = \texttt{notify(e);} \Longrightarrow s \xrightarrow{\tau}_i s'$$

  with $s'.\overline{V} = s.\overline{V}$, $s'.\overline{PC} = \nu_i(s.\overline{PC})$, $s'.\sigma_j = \mathit{runnable}$ for $j \in \mathcal{A}(I)$, and $s'.\sigma_j = s.\sigma_j$ for $j \notin \mathcal{A}(I)$.

- If I is a statement that assigns the value of the expression e to the variable x, the thread i makes a τ-transition and changes the global state accordingly. Let $s(e)$ denote the value of the expression e evaluated in state s.

  $$I = \texttt{x=e;} \Longrightarrow s \xrightarrow{\tau}_i s'$$

  with $s'.x = s(e)$, $s'.y = s.y$ for $y \neq x$, $s'.\overline{PC} = \nu_i(s.\overline{PC})$, $s'.\sigma_j = \mathit{runnable}$ for $j \in \mathcal{A}(I)$, and $s'.\sigma_j = s.\sigma_j$ for $j \notin \mathcal{A}(I)$. If the modification of x triggers events that other threads are sensitive to, this can be realized by an implicit `notify` statement after the assignment.

- If I is a guarded `goto` statement with guard g and target t, the thread i makes a τ-transition and changes its PC accordingly:

  $$I = \texttt{if(g) goto t;} \Longrightarrow s \xrightarrow{\tau}_i s'$$

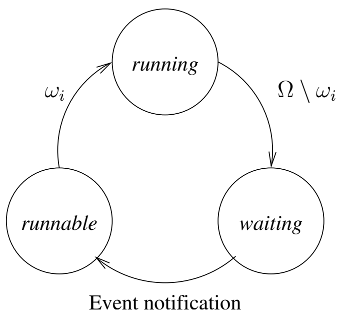

Figure 3. Transitions between the states of thread i.

  with $s'.\overline{V} = s.\overline{V}$, $s'.\sigma = s.\sigma$, and

  $$s'.PC_j = \begin{cases} t & : \quad i = j \wedge s(g) \\ PC_j + 1 & : \quad \text{otherwise} \end{cases}$$

- If I is a `wait` statement, the thread i transitions into the *waiting* state. Any of the actions in $\Omega \setminus \omega_i$ is performed during the transition in order to synchronize with any one of the other threads. This formalizes the non-deterministic choice the SystemC process scheduler makes among the runnable threads. Formally,

  $$I = \texttt{wait();} \Longrightarrow s \xrightarrow{\Omega\setminus\omega_i}_i s'$$

  with $s'.\overline{V} = s.\overline{V}$, $s'.\overline{PC} = \nu_i(s.\overline{PC})$, $s'.\sigma_i = \mathit{waiting}$. The `wait()` statement with event arguments can be formalized in a similar manner.

If i is not the running thread, the only transition LKS M_i is allowed to make is an Ω transition to synchronize with the `wait` statement of the running thread. If the action ω_i is performed, the thread transitions into the *running* state. This is only allowed to happen if thread i is actually in the *runnable* state. Formally,

$$s.\sigma_i = \mathit{runnable} \Longrightarrow s \xrightarrow{\omega_i}_i s'$$

with $s'.\sigma_i = \mathit{running}$.

If action ω_j with $j \neq i$ is performed, the thread remains in its current state, i.e.,

$$s \xrightarrow{\Omega\setminus\omega_i}_i s'$$

with $s'.\sigma_i = s.\sigma_i$.

The transitions between the states of thread i are illustrated in Figure 3.

4 Hardware/Software Partitioning

Before computing the abstraction, we try to identify hardware-like threads of the SystemC program. We then synthesize a transition relation which combines multiple instructions into one transition. We syntactically distinguish three different categories of threads: *combinational threads*, *clocked threads*, and *unrestricted threads.*

Combinational Threads Combinational threads are required to satisfy the following requirements:

1. The thread must be sensitive to all input values.

2. The thread must not be sensitive to either positive or negative edges of signals.

3. The thread must not call `wait()`.

4. The thread must not contain unbounded loops.

Note that the thread does not need to be synthesizable according to the requirements of a synthesis tool; e.g., the thread may dynamically create or destroy objects. We transform combinational threads into a formula f by running Bounded Model Checking (BMC) on it as described in [27]. The thread is then removed from the model. Whenever variables that are outputs of the combinational thread are read in another thread, we add f as the constraint.

Clocked Threads Clocked threads are required to satisfy the following:

1. The thread must be sensitive to a positive or negative edge of a designated clock signal.

2. The thread must not call `wait()` with arguments.

3. Inside any unbounded loop, the thread must call `wait()`.

Clocked threads are transformed into a state machine. We add a state for the beginning and end of the thread and each `wait()` location. The transitions between the states are the paths between the corresponding program locations. The new thread performs one transition with each step of the state machine.

Unrestricted threads are threads that are neither combinational nor clocked. They are considered to be the software part of the design, and not changed.

5 Model Checking with Abstraction

For verification of the SystemC program we first construct its abstract model. We employ the predicate abstraction method for the automated computation of the abstract models. We use LKSs for modeling of SystemC threads, and labels on the LKSs states correspond to predicates that we use for predicate abstraction.

Abstraction with SAT System-level languages make extensive use of bit-vector constructs. As done in [23], we therefore use SAT in order to compute the abstraction of SystemC programs. The SAT-based approach allows handling the bit-vector constructs. This section provides a short overview of the algorithm. For more information on the algorithm, we refer the reader to [13, 23].

Recall that S denotes the (global) set of concrete states. Let $\alpha(s)$ with $s \in S$ denote the abstraction function. The abstract model can make an A-transition from an abstract state $\hat{s}$ to $\hat{s}'$ iff there is an A-transition from s to s' in the concrete model and s is abstracted to $\hat{s}$ and s' is abstracted to $\hat{s}'$. Let $\hat{T}$ denote this abstract transition relation. Formally,

$$\hat{s} \xrightarrow{A} \hat{s}' \quad :\Longleftrightarrow \quad \begin{array}{l} \exists s, s' \in S : s \xrightarrow{A} s' \wedge \\ \alpha(s) = \hat{s} \wedge \alpha(s') = \hat{s}' \end{array} \tag{1}$$

This formula is transformed into CNF by replacing the bit-vector arithmetic operators by arithmetic circuits. Due to the quantification over the abstract states this corresponds to an all-SAT instance. For efficiency, one over-approximates $\hat{T}$ by partitioning the predicates into clusters [24]. The use of SAT for this kind of abstraction was first proposed in [11].

Thread-modular Abstraction The abstract models are built separately for each LKS corresponding to an individual SystemC thread. The advantage of this approach is that the individual threads are much smaller than the overall program. After abstracting the thread separately, we form the parallel composition of the abstract LKSs that then can be verified.

The following formalizes our modular abstraction approach.

Let M_1 and M_2 be two LKSs, and let $\pi = \langle s_1, a_1, \ldots \rangle$ be an alternating infinite sequence of states and actions of $M_1 \parallel M_2$. The *projection* $\pi \lceil M_i$ of π on M_i consists of the (possibly finite) subsequence of $\langle s_1, a_1, \ldots \rangle$ obtained by simply removing all pairs $\langle a_j, s_{j+1} \rangle$ for which $a_j \notin \Sigma_i$. In other words, we keep from π only those states that belong to M_i, and excise any transition labeled with an action not in M_i's alphabet.

We now record the following claim, which extends similar standard results for the process algebra CSP [35] and LKSs [7].

Claim 1

1. *Parallel composition is (well-defined and) associative and commutative up to $\sim$-equivalence. Thus, in particular, no bracketing is required when combining more than two LKSs.*

2. *Let $\hat{M}_i$ denote the abstraction of M_i, and let $\hat{M}_{||}$ denote the abstraction of the parallel composition of $M_1, \ldots, M_n$. Then $\hat{M}_1 || \ldots || \hat{M}_n \sim \hat{M}_{||}$. In other words, the composition of the abstract machines ($\hat{M}_1, \ldots, \hat{M}_n$) is an abstraction of the composition of the concrete machines ($M_1, \ldots, M_n$).*

The correctness of the claim follows from the fact that only one of the threads M_i can make a transition in each step. For detailed related proofs of the compositional approach, we refer the reader to [35].

Claim 1 formalizes our thread-modular approach to abstraction. Simulation and refinement can also be performed without building the transition relation of the product machine. This is justified by the fact that the program visible state variables ($\overline{V}$ and $\overline{PC}$) are only changed by one thread on shared transitions. Thus, abstraction, counterexample validation and abstraction refinement can be conducted one thread at a time.

Abstraction-refinement Loop Once the abstract model is constructed, it is passed to the model checker for the consistency check against the properties. In this project, we use the SatAbs model checker [14], which implements the SAT-based predicate abstraction approach for verification of ANSI-C programs. It employs a full counter-example guided abstraction refinement verification approach. Following the abstraction-refinement loop, this project iteratively refines the abstract model of the SystemC program if it is detected that the counterexample produced by the model checker can not be simulated on the original program. Since spurious counterexamples are caused by existential abstraction and since SAT solvers are used to construct the abstract models, we also use SAT for the simulation of the counterexamples. Our verification tool forms a SAT instance for each transition in the abstract error trace. If it is found to be unsatisfiable, it is concluded that the transition is spurious. As described in [12], the tool then uses the unsatisfiable core of the SAT instance for efficient refinement of the abstract model.

Clearly, the absence of individual spurious transitions does not guarantee that the error trace is real. Thus, our model checker forms another SAT instance. It corresponds to Bounded Model Checking (BMC) [5] on the original SystemC program following the control flow and thread schedule given by the abstract error trace. If satisfiable, our tool builds an error trace from the satisfying assignment, which shows the path to the error. A similar approach is used in [18] for DSP software. The counterexample trace includes values for all concrete variables that are assigned on the path. If unsatisfiable, the abstract model is refined by adding predicates using weakest preconditions. Again, we use the unsatisfiable core in order to select appropriate predicates.

Pointers and Dynamic Memory Allocation SystemC programs, based on C++, make frequent use of dynamically allocated objects using the `new` operator. We support such constructs by the following means:

- We allow pointers and pointer dereferencing operators within the predicates.

- For each pointer that is assigned a dynamic object, we have special predicates that keep track of the size and an active bit, which is set upon calling `new`, and cleared upon calling `delete`. Each time the pointer is dereferenced, we assert that the `active` predicate holds. We denote the predicate by $\alpha(o)$, for any object o.

- During the construction of Equation (1), we employ a standard, but control flow-sensitive points-to analysis in order to obtain the set of variables a pointer may point to. This is used to perform a case-split in order to replace the pointer dereferencing operators. Dynamic objects are handled as follows: We generate exactly as many instances as there are different points that may alias to the same dynamic object.

This approach not only allows handing pointers and pointer arithmetics of SystemC programs, but also efficiently manages the size of the generated CNF equations since it avoids handling data that pointers do not point to.

Example Figure 4 shows an example of predicate abstraction in the presence of dynamically allocated objects. The left hand side shows the code to be abstracted, the right hand side shows the predicates

```
struct s {
       s *n;
       int i;
} *p;
...
*p=new s;              α(*p)
p->n=new s;            α(*p), α(*(p->n))
p->n->i=p->i+1;        p->n->i = p->i + 1
```

Figure 4. Example of Abstraction in Presence of Dynamic Objects.

that hold after the execution of the code. In order to show the last predicate, the equality of the two integer fields, the following formula is built, where D_1 and D_2 denote the two dynamic objects, and b_3 denotes the Boolean variable corresponding to the predicate $p\text{->}n\text{->}i = p\text{->}i + 1$:

$$\begin{gathered} p = \&D_1 \wedge D_1.n = \&D2 \wedge \\ D_2'.p = D_2.p \wedge D_2'.i = D_1.i \wedge \\ (b_3 \iff (D_2'.i = D_1.i + 1)) \end{gathered}$$

This formula is only valid for $b_3 = \texttt{true}$, which shows the predicate.

6 Experimental Results

In our evaluation of the software/hardware partitioning technique, we used ZChaff [32] for the abstraction, simulation, and as Core-extractor. We used NuSMV for checking the abstract models. The experiments were performed on machine with a 2.8 GHz Intel processor with 4GB RAM.

Interrupts are a common mechanism to implement efficient data-transfer between a hardware and a software component of a system. It avoids the overhead of a polled-I/O implementation. As an example for a hardware/software Co-design, we used SystemC to model an UART together with system software that reads the data from the UART upon arrival.

The system has two modules: 1) the hardware module reads data from the environment and stores it in a register. Simultaneously, an Interrupt Service Request (IRQ) signal is raised for one cycle. 2) The interrupt service routine (ISR) is a thread of the second module. It is sensitive to the positive edge of the IRQ signal.

The property we checked was that no data that arrived at the UART was lost. We specified this property by adding a short counter to both the ISR and the UART, denoted by `sw.counter` and `uart.counter`. We verified that both counters were at most one apart.

$$\textbf{AG}\quad \begin{aligned} &\texttt{uart.counter} = \texttt{sw.counter} \vee \\ &\texttt{uart.counter} = \texttt{sw.counter} + 1 \end{aligned}$$

Without partitioning, we obtained a model with a total of 25 threads. NuSMV failed to complete on even the initial abstract model with this number of threads within the one hour time limit. This large number of threads was mainly caused by threads that model combinational logic. Upon removal of these threads using the technique described in section 4, we obtained only 2 threads. The property above could be verified with a total of 25 refinement steps in 450 seconds.

After transforming the single clocked thread as described in section 4, the number of refinement steps was reduced to 4, and the property was verified in a total runtime of 35 seconds.

7 Conclusion

This paper formalizes the semantics of SystemC by means of labeled Kripke structures (LKSs). The LKS notation labels both states and the transitions of the model. The labels on the states allow an efficient treatment of the program data, whereas the transition labels are used to model the synchronization between the threads. The SystemC program modeled as a set of LKSs is partitioned automatically into a hardware and a software part by syntactically identifying combinational and clocked threads. Combinational threads are removed, and the transitions of clocked threads are compressed, which altogether simplifies the verification problem.

Future Work As future work, we plan to perform more aggressive simplifications of the thread structure by analyzing the abstract models. Thus, the hardware/software partitioning would no longer be static, but dependent on the abstraction, and would change during the abstraction refinement.

References

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

[2] T. S. Anantharaman, E. M. Clarke, M. J. Foster, and B. Mishra. Compiling path expressions into VLSI circuits. In *Proceedings of POPL*, pages 191–204, 1985.

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

[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 *TACAS*, pages 193–207, 1999.

[6] S. Chaki, E. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in C. In *ICSE*, pages 385–395, 2003.

[7] E. Clarke, S. Chaki, N. Sharygina, J. Ouaknine, and N. Sinha. State/event-based software model checking. In *Proceedings of the International Conf. on Integrated Formal Methods, LNCS 2999*, 2004.

[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, H. Jain, and D. Kroening. Predicate Abstraction and Refinement Techniques for Verifying Verilog. Technical Report CMU-CS-04-139, Carnegie Mellon University, School of Computer Science, 2004.

[13] E. Clarke, D. Kroening, N. Sharygina, and K. Yorav. Predicate abstraction of ANSI–C programs using SAT. *Formal Methods in System Design*, 25:105–127, September–November 2004.

[14] E. Clarke, D. Kroening, N. Sharygina, and K. Yorav. SATABS: SAT-based predicated abstraction for ANSI-C. In *Proceedings of the 11th International Conf. on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, 2005.

[15] 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.

[16] M. Colón and T. Uribe. Generating finite-state abstractions of reactive systems using decision procedures. In *CAV*, volume 1427 of *LNCS*, pages 293–304. Springer, 1998.

[17] B. Cook, D. Kroening, and N. Sharygina. Accurate theorem proving for program verification. Technical Report 473, ETH Zurich, January 2005.

[18] D. W. Currie, A. J. Hu, and S. Rajan. Automatic formal verification of DSP software. In *Proceedings of DAC 2000*, pages 130–135. ACM Press, 2000.

[19] R. Drechsler and D. Große. Reachability analysis for formal verification of SystemC. In *Euromicro Symposium on Digital System Design (DSD'02)*, page 337, September 2002.

[20] S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *CAV*, volume 1254 of *LNCS*, pages 72–83. Springer, 1997.

[21] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Symposium on Principles of Programming Languages*, pages 58–70, 2002.

[22] J. Ivers and N. Sharygina. Overview of ComFoRT, a model checking reasoning framework. Technical Report CMU/SEI-2004-TN-018, CMU, April 2004.

[23] H. Jain, E. Clarke, and D. Kroening. Verification of SpecC and Verilog using predicate abstraction. In *Proceedings of MEMOCODE 2004*, pages 7–16. IEEE, 2004.

[24] H. Jain, D. Kroening, N. Sharygina, and E. Clarke. Word level predicate abstraction and refinement for verifying RTL Verilog. In *Proceedings of DAC 2005*, 2005. To appear.

[25] E. Kindler and T. Vesper. ESTL: A temporal logic for events and states. *Lecture Notes in Computer Science*, 1420:365–383, 1998.

[26] D. Kroening and E. Clarke. Checking consistency of C and Verilog using predicate abstraction and induction. In *Proceedings of ICCAD*, pages 66–72. IEEE, November 2004.

[27] 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.

[28] 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.

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

[30] R. Kurshan. *Computer-Aided Verification of Coordinating Processes.* Princeton University Press, Princeton, 1995.

[31] K. Man. Enhancing formal methods for SystemC designs. In *PROGRESS 2003 Embedded Systems Symposium*, 2003.

[32] 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.

[33] R. D. Nicola and F. Vaandrager. Three logics for branching bisimulation. *Journal of the ACM (JACM)*, 42(2):458–487, 1995.

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

[35] A. W. Roscoe. *The Theory and Practice of Concurrency.* Prentice-Hall International, London, 1997.

[36] T. Sakunkonchak and M. Fujita. Verification of event-based synchronization of SpecC description using difference decision diagrams. In D. Peled and M. Y. Vardi, editors, *Formal Techniques for Networked and Distributed Systems (FORTE 2002)*, volume 2529 of *Lecture Notes in Computer Science*. Springer, 2002.

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

[38] C. Schulz-Key, M. Winterholer, T. Schweizer, T. Kuhn, and W. Rosenstiel. Object-oriented modeling and synthesis of SystemC specifications. In *Proceedings of the Asia South Pacific Design Automation Conference (ASPDAC'04)*, pages 238–243, 2004.