Deductive Verification of Parameterized Embedded Systems Modeled in SystemC

Philip Tasche[1(✉)], Raúl E. Monti[1], Stefanie Eva Drerup[2], Pauline Blohm[2], Paula Herber[2], and Marieke Huisman[1]

[1] University of Twente, Enschede, The Netherlands
{p.b.h.tasche,r.e.monti,m.huisman}@utwente.nl
[2] University of Münster, Münster, Germany
{s.dr,pauline.blohm,paula.herber}@uni-muenster.de

Abstract. Major strengths of deductive verification include modular verification and support for functional properties and unbounded parameters. However, in embedded systems, crucial safety properties often depend on concurrent process interactions, events, and time. Such properties are global in nature and thus difficult to verify in a modular fashion. Furthermore, the execution and scheduling semantics of industrially used embedded system design languages such as SystemC are typically only informally defined. In this paper, we propose a deductive verification approach for embedded systems that are modeled with SystemC. Our main contribution is twofold: 1) We provide a formal encoding and an automated transformation of SystemC designs for verification with the VerCors deductive verifier. 2) We present a novel approach for invariant construction to abstractly capture global dependencies. Our encoding enables an automated formalization and deductive verification of parameterized SystemC designs, and the invariant construction enables local reasoning about global properties with comparatively low manual effort. We demonstrate the applicability of our approach on three parameterized case studies, including an automotive control system.

1 Introduction

Embedded systems have become ubiquitous in our daily life, including in safety-critical systems such as cars, airplanes or medical instruments. This makes systematically ensuring their correctness crucial and formal verification highly desirable. However, there are two major challenges when it comes to the formal verification of embedded systems: First, the languages that are used for embedded system design in industry are typically only informally defined, and their semantics is hard to capture formally due to their concurrency and timed behavior. Second, important properties of embedded systems, such as deadlock freedom or a timely reaction to events, typically depend on the scheduling, concurrent process interactions, events and time, and thus are global in nature. This makes it difficult to tackle them with modular verification approaches.

R. Dimitrova et al. (Eds.): VMCAI 2024, LNCS 14500, pp. 187–209, 2024.
https://doi.org/10.1007/978-3-031-50521-8_9

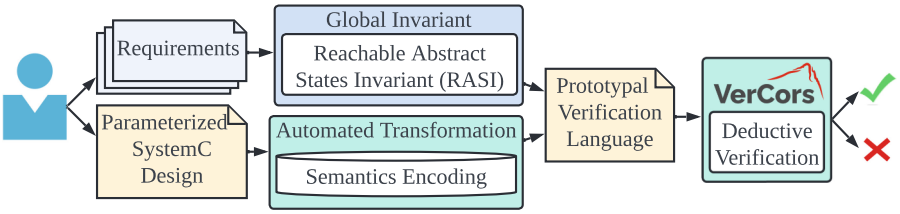

Fig. 1. Transformation-based Deductive Verification.

A language for embedded system design that is widely used in industry is SystemC [26]. In SystemC, concurrent processes are executed by a non-preemptive scheduler and are triggered by events. The execution semantics of SystemC is informally defined in [26]. To verify SystemC designs, several approaches have been proposed, e.g. [10,15,22,24,27,28]. However, these are based on model checking and are thus limited, e.g. when it comes to unbounded parameters. This is especially relevant for SystemC, since it is often used at early development stages before system parameters, like buffer sizes or threshold values, are fixed.

This paper presents a deductive verification approach for parameterized embedded systems that are modeled in SystemC. Unlike model checking, deductive verification can cope with unbounded parameters as well as functional properties by using inference rules and mathematical deduction. Furthermore, deductive verification supports modular verification using local reasoning based on method contracts. This enables compositional verification, even for partially implemented systems, and significantly increases reusability. Our goal is to use deductive verification to obtain formal and exhaustive guarantees while mitigating the state space explosion incurred by parameterized systems. Our main contribution is twofold: First, we provide a formal encoding of key aspects of the SystemC execution semantics and an automated transformation from SystemC into PVL, an input language of the VerCors deductive verifier [7]. Second, we present an approach for invariant generation that allows us to use modular verification for global properties.

Our approach is based on five key ideas. To encode the execution semantics of SystemC, we 1) provide a predefined abstract representation of the scheduler, 2) capture the event mechanism by defining shared global variables that describe event notifications and process states, and 3) define a global mutex that ensures non-preemptive execution. To use local reasoning for global properties, we 4) associate the global mutex with a global invariant that relates the states of events and processes to the desired properties, and 5) automatically generate a *Reachable Abstract States Invariant* (RASI), which enumerates reachable abstract process and event states and is included in the global invariant.

Our encoding enables us to automatically transform a given parameterized SystemC design into PVL, as shown in Fig. 1. For the automated transformation, we use our encoding of the SystemC semantics and additionally translate user-defined processes, methods, and data structures into PVL. The designer can then

formally analyze crucial safety properties with VerCors by adding specifications to the transformed program, while the RASI restricts the analysis to the relevant part of the state space. Note that, while our approach is specialized for SystemC, the core ideas can equally be applied to similar systems, such as RTOS.

We have implemented our approach and demonstrate its applicability on three parameterized case studies, including an automotive control system that could not be verified with existing approaches using, for example, the UPPAAL model checker [5,21] or the CPAchecker [6,20].

This paper is structured as follows. Section 2 introduces SystemC and VerCors. Section 3 presents our transformation from SystemC into PVL, and Sect. 4 shows how to use our encoding for verification. Section 5 shows experimental results and discusses the applicability and scalability of our approach. Section 6 summarizes related work and Sect. 7 concludes the paper.

2 Background

In this section, we introduce the necessary background for the remainder of this paper, namely SystemC and VerCors.

2.1 SystemC

systemc is a modeling language and a simulation framework for hardwaresoftware co-design. It is implemented as a `C++` library and its semantics is informally defined in an IEEE standard [26]. SystemC enables modeling and simulation at different levels of abstraction, from functional over transaction level design down to register transfer level. The SystemC library implements primitives for the design of interacting processes as well as an event-driven simulation kernel. A SystemC design consists of *modules* that are connected by *channels*. A module defines *processes* that run concurrently and interact through *events*. SystemC uses non-preemptive scheduling, i.e. a process will not be interrupted until it finishes or gives up control with a `wait` statement. Processes can wait for a given amount of time or for an event. Like typical hardware simulators, SystemC uses the concept of *delta cycles* to impose a partial order on parallel processes, i.e. concurrent processes are executed sequentially but in zero time.

The SystemC simulation kernel controls the simulation time and the execution of processes, handles event notifications and updates channels for communication. The execution semantics can be summarized as follows [26]: 1. Initialization: execute each process once; 2. Evaluation: while processes are ready to run, execute them in arbitrary order[1]; 3. Update: update channels; 4. If there are delta-delayed notifications, wake up the corresponding processes and go to step 2; 5. If there are timed notifications, advance simulation time to the earliest pending timed notification and go to step 2; 6. If there are no timed notifications remaining, simulation is finished.

[1] While implementations of the simulation semantics often use a deterministic execution order, the IEEE standard does not define any such order [26].

```
1  SC_MODULE(Robot) {
2    sc_event od;
3    void sensor() {
4      int dist = -1;
5      while(true) {
6        wait(2, SC_MS);
7        dist = read_sensor();
8        if(dist < MIN_DIST) {
9          od.notify(SC_ZERO_TIME);
10       }
11     }
12   }
13   void controller() {
14     bool flag = false;
15     while (true) {
16       wait(od);
17       flag = true;
18     }
19   }
20   SC_CTOR(Robot) {
21     SC_THREAD(sensor);
22     SC_THREAD(controller);
23   }
24 };
```

Fig. 2. Running example of a simple robot in SystemC.

Figure 2 shows a SystemC design of a simple robot, which we use as a running example in this paper. It consists of just one module, declared in line 1, with two processes `sensor` and `controller`, defined as methods and declared as thread processes in lines 21 and 22. The `sensor` process periodically reads sensor values by waiting for 2 ms (line 6) and then reading a new `dist` value (line 7). If `dist` falls below a threshold, which is given by system parameter `MIN_DIST`, the sensor notifies an obstacle detection event `od`. The event notification is delayed by one delta cycle, indicated by the `SC_ZERO_TIME` keyword. The `controller` process waits for `od` (line 16) and sets a flag (line 17) if it receives this event.

2.2 VerCors

vercors is a deductive program verifier that specializes in data race freedom, memory safety and functional correctness of concurrent programs [7]. It supports different concurrency models for programs written in Java, OpenCL, OpenMP for C and PVL, its own Prototypal Verification Language. We use PVL for our encoding because of its flexibility and support for different language features. PVL is an object-oriented language with a syntax and semantics similar to Java.

VerCors uses *contract-based reasoning*. Methods are annotated with pre- and postconditions specified in permission-based separation logic (PBSL) [3]. VerCors *automatically* verifies whether the annotated code complies with its contract in a modular way: It verifies in isolation that each method fulfils its contract and replaces the method by its contract at the call site. Pre- and post-conditions are indicated by the keywords `requires` and `ensures`, respectively.

Besides first order logic formulas, contracts in VerCors also include heap *permission* obligations. To access a heap variable, a thread must have permission to do so. An annotation `Perm(x,`π`)` with $\pi \in (0, 1]$ specifies read access permission to the heap location `x`, and write access permission if π is 1 [2]. Permissions can be split and merged as long as the total sum of permissions to a field stays constant, e.g. `Perm(x,1)` can be split into two terms `Perm(x,1\3) ** Perm(x,2\3)`, e.g. to distribute it over two threads, and later recombined into `Perm(x,1)`. Here, the operator `**` represents the separating conjunction from PBSL and indicates that resources on both sides are disjoint. VerCors uses these annotations to reason about absence of data races and memory safety in concurrent programs.

```
1  class BankAccount{
2    int balance;
3    boolean active;
4    resource lock_invariant() = Perm(balance, 1) ** balance >= 0;
5
6    ensures Perm(active, 1) ** active;
7    BankAccount(){                {= Perm(active, 1) ** Perm(balance, 1) =}
8      active = true;              {= Perm(active, 1) ** active ** Perm(balance, 1) =}
9      balance = 0;                {= ... ** Perm(balance, 1) ** balance == 0 =}
10   }                             {= Perm(active, 1) ** active =}
11
12   requires amount > 0;
13   requires Perm(active, read) ** active;
14   void deposit(int amount){     {= ... =}
15     lock(this);                 {= ... ** Perm(balance, 1) ** balance >= 0 =}
16     balance = balance + amount; {= ... ** Perm(balance, 1) ** balance > 0 =}
17     unlock(this);               {= ... =}
18   }
19 }
```

Fig. 3. Intrinsic locks and lock invariants in VerCors.

Locks are the main way to synchronize threads in PVL. They are not explicitly defined in code; rather, every object in PVL is associated with an intrinsic lock that can be acquired and released with the keywords `lock` and `unlock`. To reason about correct synchronization, a lock declares a *lock invariant*, which includes access permissions and functional properties about the shared state protected by the lock [34]. A lock invariant must first be established in the constructor. Then, when a thread acquires the lock, it also acquires all permissions and assumes all properties in the invariant. When the lock is released, the invariant must be reestablished and the permissions are returned to the lock [2].

Figure 3 illustrates the use of contracts and lock invariants in VerCors. The annotations between `{=` and `=}` are not part of the PVL program, but indicate the permissions and knowledge at a point. The `BankAccount` class implements a simple bank account that has two fields, `balance` and `active`, and allows concurrent deposits. To avoid data races, access to the `balance` field is protected by a lock by including its permission in the lock invariant (line 4). The lock invariant also specifies a functional property that ensures a nonnegative balance. Since the lock invariant must be established every time the lock is released (e.g. in line 17), this encodes a global property about `balance`. The constructor generates permissions for both fields. However, it passes the permission to `balance` to the lock (line 10) and only returns permission to `active` (line 6). The precondition of the method `deposit` (lines 12 and 13) uses this permission to require that the account is active (line 13). It requires `read` permission, which means an arbitrary amount of permission $\pi \in (0, 1]$. It also needs permission to update `balance`, which it gets from the lock (line 15). Additionally, it can assume the global invariant (`balance >= 0`) at this point. Since it requires a positive deposit (line 12) and the mathematical integers VerCors uses cannot overflow, the assignment in line 16 cannot violate the global invariant and the method is able to reestablish the lock invariant in line 17 when it releases the lock.

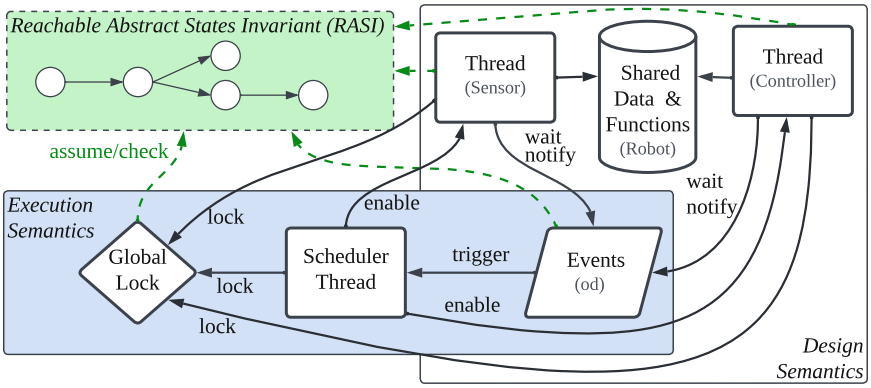

Fig. 4. Overview of the SystemC Encoding in PVL.

3 Encoding the SystemC Semantics in PVL

This section presents our formal encoding of the execution semantics of SystemC and our automated transformation into PVL. Our transformation formalizes the informal SystemC semantics and enables its deductive verification.

Figure 4 illustrates the overall structure of our encoding. Rectangular boxes indicate threads, the diamond a lock, the rhomboid a data structure for events, and the cylinder a class containing other data structures and functions. Our encoding consists of two parts: the general *Execution Semantics* of SystemC and the *Design Semantics* of a given SystemC design.

To encode the SystemC *Execution Semantics*, we define a model of the scheduler, which is run in its own *Scheduler Thread*, and we encode *Events* as globally accessible variables. The scheduler *enables* threads based on process states and event triggers. The *Global Lock* serves as a mutex to encode non-preemptive scheduling. Every thread, including the scheduler, contends on this lock. If the process that acquires the lock is *enabled*, it executes, otherwise it releases the lock again.

To encode the *Design Semantics* of a given SystemC design, we translate each process into a runnable class, instantiated in its own *Thread*, as illustrated with the *Sensor* and the *Controller* from our running example (see Fig. 2). We add the necessary variables for the events (*od* in our example), and shared data and functions (from the *Robot* module in our example). The threads can notify or wait on events and can access shared data and functions.

Figure 4 also shows the embedding of the *Reachable Abstract States Invariant* (RASI) into our encoding of the SystemC semantics. The RASI abstractly describes the reachable states by enumerating all reachable combinations of event and process states. When verifying our designs, we include the RASI in the global lock invariant to restrict the analysis to interesting states. We explain the generation of the RASI and its use in deductive verification in Sect. 4.

In the following, Sect. 3.1 defines the SystemC subset we currently support. Section 3.2 explains our encoding of the SystemC execution semantics and Sect. 3.3 describes our transformation of a given SystemC design.

3.1 Assumptions

The language subset that we support comprises key features for hardware/-software co-design and embedded systems modeling, most importantly the non-preemptive scheduling semantics, concurrent processes, dynamic event sensitivity, and time. This allows us to reason about crucial properties of embedded systems, such as deadlock freedom and a timely reaction to events. However, we still make some assumptions. As is standard among formal verification approaches for SystemC, we assume a static design without dynamic process creation, dynamic port binding or dynamic memory allocation. Without loss of generality[2], we also assume a flattened design without static sensitivity lists. Supported data types are restricted by VerCors to booleans, (mathematical) integers and arrays thereof. VerCors offers only limited support for pointers and floats, although there is ongoing research on extending this support. Currently unsupported features that we can conceptually support and that we plan to add to our automatic transformation in the future include primitive channels and multiple-event sensitivity.

3.2 Encoding of Events and Concurrent Processes

To verify a SystemC design, we need to formally encode the semantics of SystemC, in particular the event mechanism and the process scheduling. This section presents our approach to encode key aspects of the SystemC semantics.

Non-Preemptive Execution. We introduce a *global lock* to encode the non-preemptive execution of concurrent processes as established by SystemC. This lock guards the whole state of the design, ensuring that there is always a single process executing. When a process starts execution, its first statement is to lock on the *global lock*, as shown in line 2 of Fig. 5a. The process is then non-preemptively executed until it releases control with a wait statement. We encode this by continuously releasing the *global lock* and trying to reacquire it as long as the process has not been woken up due to the event, as shown in lines 8 and 9 of Fig. 5a. When the process releases the lock, other processes can execute. Whether a process is ready to run is determined by the current state of events and the scheduler. Conceptually, all processes compete for the lock, and one of them is nondeterministically chosen according to the semantics of locks in PVL. If it is ready to execute, then it resumes execution, otherwise it releases the lock and another process is chosen. This adheres to the SystemC simulation semantics [26], where processes are executed in arbitrary order, and it ensures that the requirements are verified for all possible execution orders.

[2] Static sensitivity is subsumed by dynamic sensitivity and a hierarchical design can be flattened by using, for example, prefixing.

```
1  void run {
2    lock(m);
3    while (true) {
4      m.event_state[0] = 2;
5      m.process_state[0] = 0;
6      while (m.process_state[0] != -1
7          || m.event_state[0] != -2) {
8        unlock(m);
9        lock(m);
10     }
11     dist = read_sensor();
12     if (dist < MIN_DIST) {
13       m.event_state[2] = -1;
14     }
15   }
16   unlock(m);
17 }
```

(a) Encoding of the sensor process.

```
1  while (true) {
2    lock(this);
3    immediate_wakeup();
4    reset_events_no_delta();
5    if (no_process_ready()) {
6      reset_occurred_events();
7      int d = min_advance(event_state);
8      advance_time(d);
9      wakeup_after_wait();
10     reset_all_events();
11   }
12   unlock(this);
13 }
```

(b) Encoding of the scheduler.

Fig. 5. Encoding of the scheduler and an example process.

Events. We encode events as globally accessible variables and associate each event with a global event ID. To encode triggers and event sensitivities, and to enable processes to continue their execution after a wait statement, we also associate each process with a distinguishing global ID. The state of all processes and events in the design is encoded in the integer arrays[3] `process_state` and `event_state`, to which we refer as *scheduling variables* in this paper. These arrays are indexed by the process and event ID, respectively. Position p in `process_state` holds the ID of the event that process p is waiting on, or -1 if the process is not waiting. Position e in `event_state` contains the remaining time until the occurrence of event e if it is nonnegative, -1 if the event is notified with delta delay, -2 if the event occurred in the current delta cycle and -3 if the event is not notified.

`wait` and `notify`. Figure 5a shows how we use the scheduling variables to encode event handling in our translation of the `Sensor` process from Fig. 2. The scheduling variables are attributes of a top level class `Main`, which is referenced with the variable `m`. We use this class both for globally shared variables and as the global lock. The example process starts by waiting for 2 ms. We encode timed waits by defining a dedicated timeout event (here with ID 0). The process notifies this event with the given delay and then waits on it. This is shown in lines 4 to 10 of Fig. 5a. For the event notification, the process sets the `event_state` of the timeout event to the required delay (2 in this case) in line 4. Lines 5 to 10 then encode the process waiting for the event. In line 5, it sets its own entry (ID 0) in the `process_state` array to the ID of the event it is waiting on. Then, in lines 6 to 10, it remains in a busy wait. It releases the global lock so that other processes can execute and then tries to reacquire it to continue execution. The loop condition in lines 6 and 7 makes sure the sensor can only continue

[3] For simplicity, we speak of arrays here. In our implementation, all collections have the PVL-intrinsic type `seq`, which reduces verification time.

```
1  // Context: The lock is held by the scheduler
2  requires held(this) ** scheduler_permission_invariant();
3  ensures held(this) ** scheduler_permission_invariant();
4  // The event state is not affected by this method
5  ensures event_state == \old(event_state);
6  // If any process is waiting on an event with 0 delay, it is woken up
7  ensures (\forall int i = 0 .. |process_state|;
8             (   \old(process_state[i]) >= 0
9              && \old(event_state[\old(process_state[i])]) == 0)
10        ==> (process_state[i] == -1));
11 // Any process not waiting on an event with delay 0 stays the same
12 ensures (\forall int i = 0 .. |process_state|;
13            (!(  \old(process_state[i]) >= 0
14               && \old(event_state[\old(process_state[i])]) == 0))
15        ==> (process_state[i] == \old(process_state[i])));
16 void immediate_wakeup();
```

Fig. 6. One of the abstract scheduler methods.

execution if it is woken up due to the occurrence of the event. Line 13 encodes a delta-delayed notification of the event with ID 2 (`od` in Fig. 2). To indicate the delta-delayed notification of the event, its `event_state` entry is set to -1.

The Scheduler. Figure 5b shows our model of the scheduler. The scheduler uses the scheduling variables to decide which event should occur next and which processes should be woken up. It also advances time by subtracting the due time to the next event from all event delays in the `event_state` array. Since the scheduler runs in its own thread, it needs to hold the global lock to execute, as seen in lines 2 and 12. Lines 3 and 4 wake up processes waiting on events with immediate notifications and reset these events. This encodes SystemC's evaluation phase. If there are still no processes ready in line 5, the scheduler advances to the next delta cycle. Line 6 resets the events that occurred in the previous delta cycle. Lines 7 and 8 advance time by finding the delay until the next event (0 if there is a pending delta-delayed notification) and subtracting it from all pending event notifications, before lines 9 and 10 again wake up processes waiting for all events that occur after the delay.

Note that the methods used in the scheduler model are not implemented but abstractly specified by their postconditions. This avoids additional verification effort for these methods. The structure of the abstract specifications is shown in Fig. 6 for the case of the `immediate_wakeup` method. Lines 2 and 3 ensure that the method has the required permissions to change the scheduling variables and that the scheduler is holding the global lock. Line 5 ensures that the events are not changed by this method. Lines 6 to 15 encode the method's effect on the process state. If a process is waiting for an event with an immediate notification, then it is woken up (Lines 7 to 10). If these conditions are not fulfilled, then that process's state should not be changed (Lines 12 to 15). This encoding comprehensively describes the method's behavior.

3.3 Encoding of SystemC Designs

Our encoding of the SystemC scheduling, non-preemptive execution and event mechanism captures the key concepts of the SystemC semantics. To transform a SystemC design into a PVL program, we add to this a translation of processes, modules and methods from the design. The result is a PVL program consistent with the semantics of the original SystemC design. Except for `wait` and `notify`, we map most SystemC statements simply onto their PVL equivalents. We treat ports and channels as external method calls. Our encoding of processes and modules is described in the following.

Processes. To encode a SystemC process in PVL, we create a runnable class for that process. Each such class is associated with a unique process ID and contains an attribute `m` as a reference to the global lock. It also defines a method `run` that encodes the SystemC process definition. This method starts by acquiring the global lock and ends by releasing it. Figure 5a illustrates the translation of the process body with the `sensor` process from our running example (lines 3 to 12 in Fig. 2). The SystemC code runs in an infinite loop that waits for 2 ms, reads new sensor data, and then notifies the `od` event if the given safety threshold is violated, i.e. if an obstacle is detected within the range of `MIN_DIST`. The PVL translation shown in Fig. 5a first acquires the lock in line 2 and then goes into the `while(true)` loop. It encodes waiting for 2 ms in lines 4 to 10, reads new sensor data in line 11, checks the distance in line 12 and notifies the *od* event in line 13. Finally, it releases the lock at termination.

Modules. If a module defines exactly one process, then any other functionality of the module, including methods and attributes, is also included in this runnable class. If the module defines more than one process, we generate a runnable class for each process and transform the remaining functionality of the module into another PVL class that holds the module's shared data and methods (see Fig. 4). Each class that encodes a process holds a reference to an instance of this shared module class. If there are multiple instances of a module, we duplicate all relevant classes for each instance to be able to consider dependencies between them. In our automated transformation, we use prefixing to distinguish between these classes.

4 Deductive Verification of SystemC Designs

With our encoding of the SystemC semantics, we can automatically transform a given SystemC design into a PVL representation. This enables us to use VerCors for deductive verification, i.e. we can verify that functional as well as safety properties hold for all possible input scenarios and all possible parameter values. This section explains how to specify and verify such properties. It also introduces the *Reachable Abstract States Invariant* (RASI), a technique to restrict the state space to consider and with that partially automate the verification process and lessen the burden on the user.

```
resource global_invariant() =
  ...
  // SENSOR
  ** Perm(sensor, read)
  ** sensor != null
  ** Perm(sensor.dist, write)
  ** Perm(sensor.pc, 1\2)
  ...
```

(a) Part of the global invariant

```
loop_invariant true
  ** Perm(m, 1\2)
  ** m != null
  ** held(m)
  ** m.global_invariant()
  ** m.sensor == this
  ** Perm(pc, 1\2)
  ** pc == 1
```

(b) Generated loop invariant

Fig. 7. Generated permissions for the sensor process from Fig. 2.

4.1 Property Specification and Verification

Our transformation to PVL enables us to use the concepts supported by VerCors for the specification and verification of SystemC designs. These are *method contracts*, *loop invariants* and *assertions*. Each allow the user to specify different aspects of the code. Method contracts describe the behavior of a method with a pre- and a postcondition. To allow modular verification, VerCors proves that the method, given the precondition, fulfils the postcondition, and that the precondition is fulfilled at every call site. Loop invariants describe properties that should hold throughout the execution of a loop and are proven before and after each loop iteration. Assertions describe properties that should hold at one specific point in the code. By using the appropriate specifications, we can reason about a wide variety of properties, both functional and global, that the original SystemC design should fulfil.

Generated Permissions. For VerCors to be able to guarantee memory safety of concurrent programs, a program must be annotated with appropriate permission obligations. To access a heap variable, a method must have permission to this variable; otherwise, verification will fail. We have designed our transformation such that we can already automatically generate almost all of the annotations needed for verification. We store permissions to all variables in the global invariant, which is included in local contracts and invariants as well as the global lock invariant. Since we require the global lock to execute, processes are guaranteed to have sufficient permissions without user intervention. The user is then left with only the task of adding the desired properties.

Figure 7 illustrates these generated permissions on the example of the sensor process from Fig. 2. Figure 7a shows the relevant part of the global invariant. It contains permissions to the `sensor` object itself as well as its attribute `dist`. It also contains permission to the sensor's program counter `pc` to allow other processes to read its value, which is useful for some properties. However, since `pc` is only updated by the sensor itself, the global invariant only contains half permission; the other half is retained by the process. This is shown in Fig. 7b, which displays a loop invariant in the sensor process. It ensures that the global lock is held for execution and that the global invariant is preserved, but it also

requires the remaining half permission to `sensor.pc` to allow the sensor to update its own program counter.

Declaring Properties. As an example of declaring a property about a SystemC design with our approach, take the simple robot from Fig. 2. Assume the user wants to verify that the controller process is only woken up if the sensor reads a distance below the threshold. In this case, after the transformation, the user can add the assertion `assert m.sensor.dist < m.MIN_DIST` to the controller process after the initial `wait` statement. If the verification succeeds, then the property is guaranteed to hold whenever the controller is woken up. Note, however, that this verification might require some auxiliary properties to succeed.

4.2 Reachable Abstract States Invariant

While functional properties can usually be verified locally, this is not the case for global properties. Typical safety properties for embedded systems, such as a timely reaction to external events, generally depend on the interplay between processes and on the scheduling. This requires additional information about the global state for local verification.

For deductive verification, this information comes in the form of user-defined invariants that capture the global state. Such invariants allow the verifier to exclude impossible states that might otherwise wrongly falsify the desired property, but they are often challenging to define by hand. To overcome this problem and reduce user effort, we lean on ideas from model checking. We enumerate the design's abstract state space with regard to its scheduling in the RASI to capture process and event dependencies. This provides a systematic and automatable way to capture the global system behavior for local verification.

To mitigate the state space explosion problem while capturing the state space, we construct an abstraction of the reachable state space that can be automatically generated. We abstract the program to a small subset A of program variables, replacing all other variables by arbitrary values. Section 4.3 explains the selection of these variables. The RASI then consists of the reachable states of this abstract program. Each state is represented by a conjunction of the values of the variables in A, and the RASI is represented by a disjunction of such states, as defined in the following.

Definition 1. Let p be a PVL program, V_p the set of all program variables of p, and $A \subseteq V_p$. An **abstract state** s with regard to A is a valuation that maps each $x \in A$ into $dom(x)$. The Boolean representation of an abstract state is defined as the condition formula $cond(s) := \bigwedge_{x \in A} x = s(x)$.

Definition 2. Let p be a PVL program and $A \subseteq V_p$. The **abstract program** $p|_A$ is the program that only takes assignments to the variables in A into consideration and performs a nondeterministic overapproximation for all other variables, i.e. all variables in $V_p \setminus A$ are treated as arbitrary values. Let now c be a program location in p. An abstract state s is **reachable with regard to A** at c in p if an execution of $p|_A$ exists for which $cond(s)$ holds at c.

```
1  resource global_invariant() =
2    /* >>> Permissions <<< */
3    ...
4    /* >>> Abstract state space <<< */
5    ** ( (process_state == [-1, -1] && event_state == [-3, -3, -3]
6          && sensor.pc == 0 && controller.pc == 0)
7       || (process_state == [0, -1] && event_state == [2, -3, -3]
8          && sensor.pc == 1 && controller.pc == 0)
9       || (process_state == [-1, 2] && event_state == [-3, -3, -3]
10         && sensor.pc == 0 && controller.pc == 1)
11      || ... )
12   /* >>> Establish the property <<< */
13   ** (event_state[2] > -2 ==> sensor.dist < MIN_DIST) // when sensor notifies od
14   ** (event_state[2] == -2 ==> sensor.dist < MIN_DIST); // when od occurs
```

Fig. 8. Excerpt of the RASI for the robot example.

Definition 3. Let p be a PVL program, c a program location in p and $A \subseteq V_p$. Let $S(A, c)$ be the set of all reachable states of p at c with regard to A. The **Reachable Abstract States Invariant (RASI)** at c with regard to A is $\mathcal{R}_A^c := \bigvee_{s \in S(A,c)} cond(s)$. The full RASI of p with regard to A is $\mathcal{R}_A := \bigvee_{c \in p} \mathcal{R}_A^c$.

Note that, while we usually speak of the RASI as the entire state space invariant $\mathcal{R}_A$ in this paper, it is often more useful to use an invariant $\mathcal{R}_A^c$ that only captures states that are reachable at program location c. This provides a smaller and stronger invariant than the full abstract reachable state space.

Figure 8 shows an example of the RASI for the robot example from Fig. 2 with A containing the scheduling variables `process_state` and `event_state` as well as program counters for both processes. Lines 5 to 11 contain 3 out of 8 reachable abstract states regarding A. Lines 5 and 6 represent the initial state, where both the sensor (process ID 0) and the controller (process ID 1) are enabled and no event has been notified or occurred yet. This state has two possible successors: either the sensor runs first and reaches the point where it waits for 2 ms (lines 7 and 8), or the controller runs first and waits for the obstacle detected event (event ID 2, lines 9 and 10).

4.3 Variables in the Abstract State

In principle, the RASI can be based on any set A of variables. To capture global inter-process dependencies, we always include the scheduling variables `process_state` and `event_state`. We also include program counters for each process in the design to capture the execution state of other processes. Data variables should generally be avoided to mitigate the state space explosion problem.

However, in some cases, the smaller abstract state can be outweighed by an increase in the potential behavior of $p|_A$. Whenever the program branches on a variable that is not in A, $p|_A$ instead performs a nondeterministic overapproximation. If the branches of the nondeterministic choice influence the variables in A, then this introduces spurious states in the RASI. For example, a buffer read operation might branch on the size of the buffer and wait if the buffer is empty. In this case, a data variable influences the process and event states. Abstracting

from the buffer size causes an overapproximation of the state space, which negatively affects verification time. In such a situation, it can be beneficial to include the concerned data variable in A as well. On the other hand, this can also add new states to the RASI any time this variable changes. The decision whether to include a variable is nontrivial in the general case, but often easy to see for a domain expert. For now, we leave it to the user.

4.4 RASI Generation and Soundness

We have implemented two RASI generators, based on different abstraction methods, which are included in our artifact [1]. The first generator is based on simulation. An abstract design is repeatedly executed while recording all visited states. This approach is conceptually simple to use, but it may lead to an underapproximation of the state space and fail the verification with VerCors.

The second RASI generator is based on systematic state space exploration. The user provides transition systems for the processes in the given design, abstracted to A. The tool then systematically explores all reachable states of the composition of these transition systems. This approach is more flexible than simulation, since it explores the abstract $p|_A$. We currently require the user to provide the necessary abstractions for each process, but we plan to automate this with existing approaches such as abstract interpretation [11] or predicate abstraction [4] in the future.

The reason that we need to impose this additional effort of abstracting the program to the variables in A is that VerCors overapproximates variables about which it has no information. An invariant that does not use the same overapproximation would not be able to be verified. This means that the state space can only depend on variables that are also included in each state's condition.

While this reasoning of VerCors creates some challenges when it comes to generating a useful RASI, it also directly implies its soundness. We include the RASI in the global invariant, and local subsets of the RASI in loop invariants and contracts. Since these are included in proof obligations for VerCors, a successful verification automatically guarantees that the RASI contains all reachable program states. If the program would enter a state that is not contained in the RASI, the verification would fail.

4.5 Verifying Global Properties with the RASI

The RASI is useful for restricting the state space under consideration and avoiding spurious verification failures for global properties. It also makes verification significantly easier for multiple classes of global properties.

Cross-Process Properties. Properties that are established in one process and need to be proven in another are difficult to prove deductively. For example, in the robot design from Fig. 2, the sensor only notifies the controller process if the value `sensor.dist` that it observes is below the threshold `MIN_DIST`. However,

proving `sensor.dist < MIN_DIST` in the controller once it is woken up requires complex global invariants that capture the dependencies between the two processes. These invariants are hard to come up with and are sensitive to changes in the system, limiting reusability. The RASI allows an easy and reusable verification technique by following the desired property from the point at which it is established to the point at which it is proven, as shown in lines 13 and 14 in Fig. 8. If the system changes, e.g. if the controller also waits before trying to prove the property (see *robot-1MS* example in Sect. 5), the user can add similar invariants for this new event with little effort.

Timing Properties. Timing constraints are usually very difficult to conceptualize for deductive verification. However, as the RASI contains timing information for events, it can help with the formalization and verification of such properties. By adding a new event, notifying this event with the desired delay to start measuring time, and checking that it has not yet occurred at the goal location, a timing property can be verified with little manual effort.

Unreachability Properties. Properties that exclude a certain abstract state, e.g. a deadlock state in which no process is ready to execute and no event is notified, are direct corollaries from the RASI. If the RASI does not contain such a state and is verified, then the program can never reach that state. Note that reachability cannot be guaranteed in this way, since the RASI might overapproximate the program behavior.

5 Evaluation

We have implemented our transformation from SystemC to PVL on top of the STATE tool [23]. To demonstrate the applicability of our approach, we conducted case studies on three example designs and some variants of them. Our tools and experiments, along with their reproduction instructions, can be found at [1]. For all three case studies, we used our transformation from SystemC to PVL to generate a formal representation. We automatically generated a RASI for an appropriate variable subset and manually added properties that capture the system requirements to the global invariant. Finally, we used VerCors to deductively verify the properties.

5.1 Case Studies

We evaluate our approach using three parameterized SystemC designs as case studies, for each of which we verify the RASI and some sample properties. Two of the case studies are smaller examples, showcasing different features of SystemC, namely our running example of a simple robot and a system of a producer and a consumer communicating via a FIFO channel. The third case study is a functional model of an automotive control system, containing an anti-slip regulation (ASR) and anti-lock braking system (ABS). It is a slightly modified

version of a case study that was developed by a student according to the Bosch specification [37]. Existing verification using, for example, the UPPAAL model checker [5,19,21] or the CPAchecker [6,20] were not able to verify this case study, and UPPAAL runs out of 64 GB of main memory both for the original and for our modified version. This is caused by the data-dependence of the properties we prove together with the missing data abstractions in UPPAAL. Even if we combine UPPAAL with an abstract interpretation as in [21], data-dependent behavior has to be unfolded in UPPAAL to avoid spurious counterexamples.

Simple Robot. The first case study is our running example of a parameterized simple robot (see Fig. 2). This case study showcases event-driven communication between processes. We verify the property that, if the controller sets the flag, then the sensor must have sensed an obstacle closer than the symbolic system parameter `MIN_DIST`. We consider several variations of the *robot* case study.

In *robot-1MS*, the controller waits 1 ms before setting the flag. This complicates the proof of the property, as it adds the requirement that the value of `dist` is unchanged during the controller's waiting period. However, since the controller waits for a shorter time than the sensor, the property still holds.

For the *robot-dummy* variant, we introduce an extra dummy process. As this dummy process does not interfere with the sensor or the controller, the property should still be verifiable.

Producer-Consumer. The second case study is a system where a producer and a consumer communicate via a FIFO channel, showcasing communication through a user-defined channel. This example has also been used in several existing works on the formal verification of SystemC, e.g. [10,14,25,43]. However, these works could only analyze the example with a fixed buffer size. In [18], the verification is performed for a variety of different fixed buffer sizes, but verification time exponentially increases for larger buffer sizes. We use a parameterized version of the producer-consumer example where the buffer size is set arbitrarily by a parameter. On this parametric design, we verify two properties. We prove that the FIFO channel, which is implemented as a circular buffer, works as expected. We further prove the global property that all items read by the consumer were sent by the producer.

ABS/ASR. Our third case study is an anti-lock braking system and anti-slip regulation (ABS/ASR) design. It contains four *tick counter* modules that measure the speed of each wheel and send it to a central *electronic control unit (ECU)* that gathers the inputs, estimates wheel and vehicle speed and acceleration, and executes the ABS or ASR algorithm. The communication takes place via FIFO channels that a dedicated process in the ECU reads from. Another ECU process reacts to the input by periodically adjusting braking pressure at the wheels. Figure 9 shows the architecture of the design with 6 processes communicating through 4 channels.

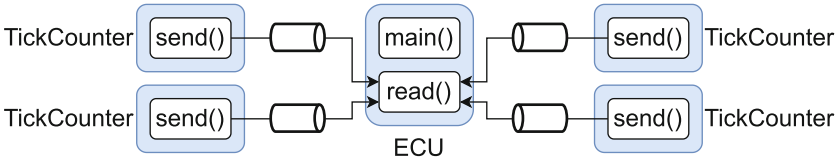

Fig. 9. Architecture of the ABS/ASR design.

We prove several properties that are crucial for this kind of automotive control system. First, we leverage the strengths of deductive verification by proving several functional properties about the correctness of the ABS and ASR algorithms. We prove that the estimation of the vehicle speed works as expected, that ABS and ASR calculate the correct slack on each wheel, that the ASR behaves correctly depending on the slack and that the ABS follows the state machine representation defined in the Bosch specification [37]. Previous work that uses model checking struggles with such properties [19,21]. Second, we prove some global safety properties, namely that the system is deadlock-free and that the ECU receives the correct data from the sensors in less than 1 ms.

To evaluate the scalability of our approach for verifying global properties, we additionally consider a variant of the ABS/ASR example in which we verify only local properties. This variant is equal to the base ABS/ASR case study except that it does not contain the global property specifications and the manual and generated invariants needed to verify them.

5.2 Experimental Results

Our experimental results are summarized in Table 1. We characterize the complexity of each of our case studies by the number of processes and events they contain, the number of methods that were generated in the PVL encoding and the number of properties we verify about them. The number of manual invariants necessary for verification and the number of states in the RASI give an indication of the verification effort. The RASI is generally based on scheduling variables and program counters; only the ABS/ASR case study also includes FIFO queue sizes to limit the overapproximation of the state space. Each case study was successfully verified in the given time, running on an i7-8700 6-core 3.2 GHz CPU with 64 GB of main memory available.

5.3 Scalability

Being able to deal with parameterized systems at all is a significant advantage of our approach over alternatives based on model checking. In addition, deductive verification is a promising technique to improve the scalability of SystemC verification. Model checking suffers from the state space explosion problem, limiting its applicability to complex designs. Due to its modular nature and local

Table 1. Experimental results.

Design	# procs	# events	# methods (in PVL)	# props	# man. invariants	RASI size $\lvert\mathcal{R}_A\rvert$	Verif. time (sec.)
robot	2	3	6	1	2	8	13
robot-1MS	2	3	6	1	4	12	15
robot-dummy	3	4	8	1	4	24	34
producer-consumer	2	4	11	2	8	72	52
ABS/ASR (local)	6	17	22	4	n.a	n.a	167
ABS/ASR	6	17	22	13	9	247*	10400

* Manually reduced with data variables, see Sect. 4.3

reasoning, deductive verification does not generally share this problem, making it a good candidate for working towards a more scalable solution.

The problem with the scalability of a deductive verification approach is the inclusion of the global state for local reasoning. In general, relating the local state to the global state is associated with high manual effort for the user. While an efficient encoding of such dependencies can lead to comparatively low verification times, the time and expertise required on the user's side often outweighs this advantage. In this work, we therefore opted for an assisted approach that would minimize user effort while maintaining some of the potential advantage in scalability that deductive verification offers. The RASI's state space enumeration allows verification of several common property classes with minimal user effort. The downside of this approach is that the RASI construction is closely related to model checking and may, therefore, suffer from the state space explosion problem.

This issue is illustrated in Table 1. While all the smaller case studies were successfully verified in less than a minute, the larger ABS/ASR case study took almost three hours to verify. A portion of this time discrepancy can be explained by the additional methods, processes and events that need to be considered and the higher number of properties that we verify about this design. However, the RASI that, even after a simple manual reduction, is still significantly larger than in the other cases[4], has the largest effect on this verification time by far. This can be seen in the comparison with the local variant of the ABS/ASR example. A potential explosion of the RASI size is a major problem for our approach.

Even so, our approach still improves the scalability of SystemC verification over model checking approaches, as is evidenced by our ability to verify the ABS/ASR example and the inability of UPPAAL and CPAChecker to do the same. The RASI, as we use it, does suffer from the state space explosion prob-

[4] Note also that, despite not necessarily increasing the state space, more processes and events also mean that the representation of each state in the RASI will have more atomic conditions.

lem, but its abstract representation mitigates this effect. Furthermore, the level of abstraction of the RASI can be controlled by the user. This makes this approach promising for a more scalable verification technique that might not be affected by state space explosion. If a property can be proven with knowledge about only a fraction of the global state, this would allow the RASI to abstract from other factors that could induce a state space explosion. While finding the right abstraction level for this is nontrivial, our approach enables such potential advances. We plan to investigate this in future work.

5.4 Discussion

Our experimental results demonstrate that our approach is applicable to the verification of parameterized SystemC designs. We were able to verify both functional and global safety properties of our case studies, including the relatively complex ABS/ASR design. In contrast to existing work based on model checking, we can verify the designs for all possible parameter values with reasonable effort with our deductive approach. This is a major advantage, in particular as SystemC is often used for design space exploration and as a golden reference model, such that concrete implementation details are fixed later in the development process. Our deductive approach also allows us to reason about unbounded data, which model checkers cannot deal with. This is vital to proving functional properties about reactive systems.

The experiments also show how the RASI can reduce user effort. While functional properties did not pose a problem, many global properties proved challenging to verify by hand. With the RASI and the verification techniques described in Sect. 4.5, it took relatively little manual effort to verify them. We could also reuse some invariants, e.g. in the *robot* case study. In our artifact documentation [1], we elaborate on the manual steps that are needed for our approach.

The main challenge proved to be the engineering of the RASI to avoid an explosion of the abstract state space. As the ABS/ASR shows, although we only take a small part of the full state space into consideration, verification time can grow quickly if the abstract state space gets too large. However, we expect it to stay manageable with careful selection of the abstract state variables as long as process interleavings are limited. Since embedded system designers try to maintain predictability in their designs, this is typically the case.

Overall, our results demonstrate the ability of our approach to verify crucial properties of parameterized embedded systems. By leveraging the strengths of deductive verification in combination with ideas from model checking, we can verify a wide range of properties with reduced user effort.

6 Related Work

There exist several approaches to enable formal verification for SystemC. Some of them only cope with a synchronous subset (e.g. [13,38,39]), others rely on a transformation of SystemC designs into some sort of state machine, as

done in [16–19,23,27,33,41,43], or they use process algebras, petri-nets or a C representation for the verification of SystemC designs [8–10,12,20,28,32]. In [24,25,30], the authors present an approach to verify SystemC designs using symbolic simulation and partial order reduction. In [14,15,27,40], the authors present a combination of induction and bounded model checking to formally verify SystemC designs. However, all of these approaches rely on model checking at some point. They cannot deal with parameterized systems, and they suffer from the state space explosion problem. In [29], the authors present an approach for automatic hardware/software partitioning, which enables the underlying verification method to analyze hardware parts more efficiently. However, they still suffer from state space explosion for the software part. In [31,42], the authors present an approach for component-based hardware/software co-verification. However, the approach requires that the designer specifies subsystems and properties that can be separately verified. In contrast to this, we present an encoding that precisely captures the SystemC execution semantics and enables us to map SystemC designs into a formal representation fully automatically. Furthermore, with our deductive approach, we gain access to the mature and sophisticated verification capabilities of VerCors. We also extend the applicability of VerCors to embedded systems that are modeled in SystemC, and gain new insights on how to encode reactivity and discrete-event mechanisms in VerCors.

7 Conclusion

In this paper, we have presented a novel approach for the deductive verification of embedded systems that are modeled in SystemC. We have proposed a formal encoding of SystemC designs in PVL that is consistent with the SystemC semantics informally defined in [26]. With that, we can use the VerCors deductive verifier to verify these designs. Our main contributions are twofold: First, we have presented an encoding of the SystemC semantics that enables an automated transformation of a given SystemC design into PVL. This spares designers the tedious task of formalizing a system themselves. Second, we have presented an approach to generate an abstract enumeration of the reachable state space with respect to process scheduling (the RASI). Both contributions are supported by the tools in our artifact [1]. Our results show that this approach allows us to deductively verify parameterized SystemC designs with comparatively low manual effort, as the RASI provides a simple and automatable technique to connect global properties with the local state.

In future work, we plan to expand the subset of SystemC we support with our automated transformation, e.g. with dynamic process creation and dynamic memory allocation. We want to automate many of the manual steps that are currently necessary, if possible, and increase the scalability of the RASI, which is currently the greatest bottleneck. We also plan to investigate the use of the VerCors-built-in support for hybrid verification [35,36] to verify global properties by model checking and then use them locally for deductive verification.

References

1. [Artifact] Deductive Verification of Parameterized Embedded Systems modeled in SystemC. https://doi.org/10.4121/a7e780c9-87fa-486c-b484-a76a459a9d53
2. Amighi, A., Blom, S., Darabi, S., Huisman, M., Mostowski, W., Zaharieva-Stojanovski, M.: Verification of concurrent systems with VerCors. In: Bernardo, M., Damiani, F., Hähnle, R., Johnsen, E.B., Schaefer, I. (eds.) SFM 2014. LNCS, vol. 8483, pp. 172–216. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-07317-0_5
3. Amighi, A., Haack, C., Huisman, M., Hurlin, C.: Permission-based separation logic for multithreaded Java programs. Log. Methods Comput. Sci. **11**(1) (2015). https://doi.org/10.2168/LMCS-11(1:2)2015
4. Ball, T., Majumdar, R., Millstein, T., Rajamani, S.K.: Automatic predicate abstraction of c programs. In: Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation, pp. 203–213 (2001)
5. Behrmann, G., David, A., Larsen, K.G.: A tutorial on Uppaal. In: Bernardo, M., Corradini, F. (eds.) SFM-RT 2004. LNCS, vol. 3185, pp. 200–236. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-30080-9_7
6. Beyer, D., Keremoglu, M.E.: CPAchecker: a tool for configurable software verification. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 184–190. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_16
7. Blom, S., Darabi, S., Huisman, M., Oortwijn, W.: The VerCors tool set: verification of parallel and concurrent software. In: Polikarpova, N., Schneider, S. (eds.) IFM 2017. LNCS, vol. 10510, pp. 102–110. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66845-1_7
8. Cimatti, A., Micheli, A., Narasamdya, I., Roveri, M.: Verifying SystemC: a software model checking approach. In: Formal Methods in Computer-Aided Design (FMCAD), pp. 51–59. IEEE (2010). https://dl.acm.org/doi/10.5555/1998496.1998510
9. Cimatti, A., Griggio, A., Micheli, A., Narasamdya, I., Roveri, M.: Kratos – a software model checker for SystemC. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 310–316. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_24
10. Cimatti, A., Narasamdya, I., Roveri, M.: Software model checking SystemC. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **32**(5), 774–787 (2013). https://doi.org/10.1109/TCAD.2012.2232351
11. Cousot, P.: Abstract interpretation. ACM Comput. Surv. (CSUR) **28**(2), 324–328 (1996)
12. Garavel, H., Helmstetter, C., Ponsini, O., Serwe, W.: Verification of an industrial SystemC/TLM model using LOTOS and CADP. In: IEEE/ACM International Conference on Formal Methods and Models for Co-design (MEMOCODE '09), pp. 46–55 (2009). https://doi.org/10.1109/MEMCOD.2009.5185377
13. Große, D., Kühne, U., Drechsler, R.: HW/SW co-verification of embedded systems using bounded model checking. In: Great Lakes Symposium on VLSI, pp. 43–48. ACM Press (2006). https://doi.org/10.1145/1127908.1127920
14. Große, D., Le, H.M., Drechsler, R.: Proving transaction and system-level properties of untimed SystemC TLM designs. In: MEMOCODE, pp. 113–122. IEEE (2010). https://doi.org/10.1109/MEMCOD.2010.5558643

15. Große, D., Le, H.M., Drechsler, R.: Formal verification of SystemC-based cyber components. In: Jeschke, S., Brecher, C., Song, H., Rawat, D.B. (eds.) Industrial Internet of Things. SSWT, pp. 137–167. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-42559-7_6
16. Habibi, A., Moinudeen, H., Tahar, S.: Generating finite state machines from SystemC. In: Design, Automation and Test in Europe, pp. 76–81. IEEE (2006). https://doi.org/10.1109/DATE.2006.243777
17. Habibi, A., Tahar, S.: An approach for the verification of SystemC designs using AsmL. In: Peled, D.A., Tsay, Y.-K. (eds.) ATVA 2005. LNCS, vol. 3707, pp. 69–83. Springer, Heidelberg (2005). https://doi.org/10.1007/11562948_8
18. Herber, P., Fellmuth, J., Glesner, S.: Model checking SystemC designs using timed automata. In: International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS), pp. 131–136. ACM Press (2008). https://doi.org/10.1145/1450135.1450166
19. Herber, P., Glesner, S.: A HW/SW co-verification framework for SystemC. ACM Trans. Embed. Comput. Syst. (TECS) **12**(1s), 1–23 (2013). https://doi.org/10.1145/2435227.2435257
20. Herber, P., Hünnemeyer, B.: Formal verification of SystemC designs using the BLAST software model checker. In: ACESMB@ MoDELS, pp. 44–53 (2014). https://dblp.org/rec/conf/models/HerberH14
21. Herber, P., Liebrenz, T.: Dependence analysis and automated partitioning for scalable formal analysis of SystemC designs. In: 18th ACM-IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE), pp. 1–6. IEEE (2020). https://doi.org/10.1109/MEMOCODE51338.2020.9314998
22. Herber, P., Liebrenz, T., Adelt, J.: Combining forces: how to formally verify informally defined embedded systems. In: Huisman, M., Păsăreanu, C., Zhan, N. (eds.) FM 2021. LNCS, vol. 13047, pp. 3–22. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-90870-6_1
23. Herber, P., Pockrandt, M., Glesner, S.: STATE - a SystemC to timed automata transformation engine. In: ICESS. IEEE (2015). https://doi.org/10.1109/HPCC-CSS-ICESS.2015.188
24. Herdt, V., Große, D., Drechsler, R.: Formal verification of SystemC-based designs using symbolic simulation. In: Enhanced Virtual Prototyping, pp. 59–117. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-54828-5_4
25. Herdt, V., Le, H.M., Große, D., Drechsler, R.: Verifying SystemC using intermediate verification language and stateful symbolic simulation. IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. **38**(7), 1359–1372 (2018). https://doi.org/10.1109/TCAD.2018.2846638
26. IEEE Standards Association: IEEE Std. 1666-2011, Open SystemC Language Reference Manual. IEEE Press (2011). https://doi.org/10.1109/IEEESTD.2012.6134619
27. Jaß, L., Herber, P.: Bit-precise formal verification for SystemC using satisfiability modulo theories solving. In: Götz, M., Schirner, G., Wehrmeister, M.A., Al Faruque, M.A., Rettberg, A. (eds.) IESS 2015. IAICT, vol. 523, pp. 51–63. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-90023-0_5
28. Karlsson, D., Eles, P., Peng, Z.: Formal verification of SystemC designs using a Petri-Net based Representation. In: Design, Automation and Test in Europe (DATE), pp. 1228–1233. IEEE Press (2006). https://doi.org/10.1109/DATE.2006.244076

29. Kroening, D., Sharygina, N.: Formal verification of SystemC by automatic hardware/software partitioning. In: Proceedings of MEMOCODE 2005, pp. 101–110. IEEE (2005). https://doi.org/10.1109/MEMCOD.2005.1487900
30. Le, H.M., Große, D., Herdt, V., Drechsler, R.: Verifying SystemC using an intermediate verification language and symbolic simulation. In: 2013 50th ACM/EDAC/IEEE Design Automation Conference (DAC), pp. 1–6. IEEE (2013). https://doi.org/10.1145/2463209.2488877
31. Li, J., Sun, X., Xie, F., Song, X.: Component-based abstraction and refinement. In: Mei, H. (ed.) ICSR 2008. LNCS, vol. 5030, pp. 39–51. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-68073-4_4
32. Man, K.L., Fedeli, A., Mercaldi, M., Boubekeur, M., Schellekens, M.: SC2SCFL: automated SystemC to $SystemC^{\mathbb{FL}}$ translation. In: Vassiliadis, S., Bereković, M., Hämäläinen, T.D. (eds.) SAMOS 2007. LNCS, vol. 4599, pp. 34–45. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-73625-7_6
33. Niemann, B., Haubelt, C.: Formalizing TLM with communicating state machines. Forum Specification Des. Lang. (2006). https://doi.org/10.1007/978-1-4020-6149-3_14
34. O'Hearn, P.W.: Resources, concurrency and local reasoning. Theor. Comput. Sci. **375**(1–3), 271–307 (2007). https://doi.org/10.1016/j.tcs.2006.12.035
35. Oortwijn, W.: Deductive techniques for model-based concurrency verification. Ph.D. thesis, University of Twente, Netherlands, December 2019. https://doi.org/10.3990/1.9789036548984
36. Oortwijn, W., Gurov, D., Huisman, M.: Practical abstractions for automated verification of shared-memory concurrency. In: Beyer, D., Zufferey, D. (eds.) VMCAI 2020. LNCS, vol. 11990, pp. 401–425. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-39322-9_19
37. Reif, K.: Bremsen und Bremsregelsysteme. Bosch Fachinformation Automobil, Vieweg+Teubner Verlag Wiesbaden (2010). https://doi.org/10.1007/978-3-8348-9714-5
38. Ruf, J., Hoffmann, D.W., Gerlach, J., Kropf, T., Rosenstiel, W., Müller, W.: The simulation semantics of SystemC. In: Design, Automation and Test in Europe, pp. 64–70. IEEE Press (2001). https://doi.org/10.1109/DATE.2001.915002
39. Salem, A.: Formal semantics of synchronous SystemC. In: Design, Automation and Test in Europe (DATE), pp. 10376–10381. IEEE Computer Society (2003). https://doi.org/10.1109/DATE.2003.1253637
40. Schwan, S., Herber, P.: Optimized hardware/software co-verification using the UCLID satisfiability modulo theory solver. In: 29th IEEE International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises, WETICE 2020, Virtual Event, France, 10–13 September 2020, pp. 225–230. IEEE (2020). https://doi.org/10.1109/WETICE49692.2020.00051
41. Traulsen, C., Cornet, J., Moy, M., Maraninchi, F.: A SystemC/TLM semantics in Promela and its possible applications. In: Bošnački, D., Edelkamp, S. (eds.) SPIN 2007. LNCS, vol. 4595, pp. 204–222. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-73370-6_14
42. Xie, F., Yang, G., Song, X.: Component-based hardware/software co-verification for building trustworthy embedded systems. J. Syst. Softw. **80**(5), 643–654 (2007). https://doi.org/10.1016/j.jss.2006.08.015
43. Zhang, Y., Vedrine, F., Monsuez, B.: SystemC waiting-state automata. In: International Workshop on Verification and Evaluation of Computer and Communication Systems (2007). https://dl.acm.org/doi/abs/10.5555/2227445.2227453