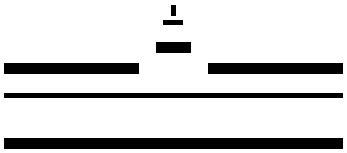
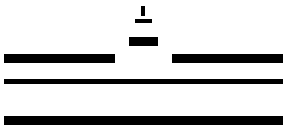


Deductive Verification of Integrated Hardware/Software Systems with the VerCors Verification Tool

Master Thesis

Stefanie Eva Drerup
– 2021 –



Deductive Verification of Integrated Hardware/Software Systems with the VerCors Verification Tool

Master Thesis
submitted in partial fulfillment of the requirements for the degree of
Master of Science in Computer Science

Computer Science Department
Faculty of Mathematics and Computer Science
University of Münster

Submitted by:
Stefanie Eva Drerup

September 30, 2021

Münster, September 30, 2021

First Reviewer and Supervisor:

Prof. Dr. Paula Herber
Embedded Systems Group
University of Münster
Münster, DE

Second Supervisor:

Dr. Raúl E. Monti
Formal Methods and Tools Group
University of Twente
Enschede, NL

Second Reviewer:

Prof. Dr. Marieke Huisman
Formal Methods and Tools Group
University of Twente
Enschede, NL

Abstract

Embedded systems are involved in nearly every part of our everyday life, for instance, in the automotive industry or the health care sector. Usually, they consist of deeply intertwinted hardware/software components, and their correctness is crucially required. Otherwise not only financial or time losses are possible, but also human lifes can be endangered.
A widely used modeling language for design and simulation of complex hardware/software systems is SystemC. It provides the full functionality of C++, extended by hardware support. SystemC supports actor-oriented design and offers the possibility of early hardware/software co-design. This enables the early verification of SystemC models during their design phase. Though, a difficulty concerning the verification of SystemC designs are their informal semantics. There have been multiple approaches in the past to give formal semantics to SystemC designs. Most of them translate the SystemC design into a formal model, which is verified by a model checking tool. Although model checking is a mathematically sound verification technique, it suffers from the state space explosion problem. Therefore, all of the different model checking approaches for the verification of SystemC designs share the problem of limited scalability.
To overcome this challenge, we investigate deductive verification techniques in this thesis. The VerCors Verifier is a tool for static verification of parallel programs. Its deductive verification is contract-based and is performed locally. Therefore, the modular verification approach of VerCors scales nicely with increasing size of the verified program. VerCors has its own Prototypal Verification Language (PVL), which has to be annotated with permissions and further specifications in order to verify the program.
In this work, we investigate a semantics-preserving transformation from SystemC models to PVL programs, which precisely captures SystemC constructs. We present transformation rules for an initial subset of SystemC language constructs. A major challenge is to preserve the non-preemptive scheduling schemantics of SystemC designs. We achieve this by providing a global locking mechanism. Although the verification with VerCors requires a significant effort to specify permissions explicitly, we identify a set of specifications that could potentially be added automatically to the transformation process.
We demonstrate our transformation rules with a small SystemC design example. Furthermore, we show how safety properties can be formalized, and verified by VerCors. We exemplify this by an one-producer-one-consumer case study modeled in SystemC and prove that no elements written to a shared FIFO are lost.

For me.

Acknowledgements

This master thesis has been a long journey for me with lots of ups and downs. I would like to thank the people in my life, without which this thesis has not been possible.
Ganz besonderer Dank gilt meiner Betreuerin **Prof. Dr. Paula Herber**. Sie hat mich immer wieder neu motiviert. Bei jedem Problem, bei jeder Unklarheit oder auch einfach Momenten der Verzweiflung, hat sie mich immer in außerordentlichem Maße unterstützt. Liebe Paula, vielen Dank für deine Menschlichkeit und deine sehr gute Betreuung.
Furthermore, I would like to thank **Prof. Dr. Marieke Huisman**. Although I felt that the progresses of my research work have always been too slow, she has been very nice and patient, and always offered great support. Thank you!
Dr. Raúl E. Monti has been an absolutely great collegue within the SAVES project. He always supported me with my questions and VerCors problems, not judging how trivial they have been. In particular, he supported me a lot during the last days before the thesis submit and comforted my last-minute-desperations. Muchas gracias por todo!
Generally, I would like to thank the **Formel Methods and Tools Group** of the UT, which helped me a lot with my VerCors problems.
In den letzten Jahren bis hin zum heutigen Tag stand mir meine gesamte Familie in sehr schwierigen Zeiten bei. Trotzdem haben meine **Eltern, Geschwister und Anhänge** immer an mich geglaubt und daran, dass ich diese Masterarbeit schreiben werde. Danke für euren Rückhalt und euer Vertrauen in mich. Ohne euch hätte ich das nicht geschafft.
Dr. Raphael Richter ist nicht nur mein Chef, sondern auch mein Mentor und Freund geworden. Raphael hat schon lange vor mir an mein Potential geglaubt und mir immer den Rücken gestärkt. So etwas hätte Meredith Grey auch getan. Vielen Dank für alles.
Ein weiteres Dankeschön geht an **Lea Föcke, Jannes Delicaris und Jannes Bantje**, auf deren LaTex-Templates diese Arbeit teilweise basiert. Außerdem haben sie so einige lästige LaTex-Fragen von mir beantwortet. Danke für die Geduld!
Ein riesiges Dankeschön geht an **Pauline Blohm, Lisa Willemsen, Phil Steinhorst, Jana Seep und Helen Möllering** für das Korrekturlesen dieser Arbeit und unfassbar wertvolles Feedback. Ganz besonders ohne die Gievenbecker Spazierrunde hätte ich wahrscheinlich die Masterarbeit doch zu den Schafen über den Zaun geworfen.
Der Wichtigste zum Schluss: Mein Freund **Marius**, der mir einfach Halt gegeben hat. Immer. Mein Fels in der Brandung. Du hast jegliche Verzweiflung und Wut meinerseits in der letzten Masterarbeitsphase einfach ertragen und trotzdem darauf beharrt, dass ich diese Masterarbeit easy schaffe. Vielen, lieben Dank an dich!

Contents

1 Introduction

Today, embedded systems have become an essential part of everyday life, for instance, in the context of *Industry 4.0*, in the automotive industry, or the health care sector. A crucial requirement is the correctness of these integrated hardware/software systems, which is hard to show due to their extensiveness and high complexity. While tests and simulations can detect errors in programs, both cannot guarantee their complete absence. The famous computer scientist Edsger W. Dijkstra mentioned this problem decades ago:

> *"Program testing can be a very effective way to show the presence of bugs, but is hopelessly inadequate for showing their absence."*
>
> — Edsger W. Dijkstra[1]

A promising approach to show properties like correctness, safety, reliability, or liveness of embedded systems is formal verification. It comprises a vast field of different techniques which are commonly based on formal methods of mathematics and logic. They range from model checking, static analysis, type systems up to deductive verification, such as theorem proving, and much more. A brief introduction to different verification techniques can be found in the guide to formal methods by O'Regan [ORe17].

1.1 Problem of Verifying Embedded Systems

The verification of integrated hardware/software systems is even more challenging than the verification of pure software or pure hardware systems, due to their increased complexity. They interact with their environment and deal with concurrency as well as real-time execution. At the same time, integrated hardware/software systems have to fulfill strict limitations of time, power and memory usage.

[1] Edsger W. Dijkstra. "The Humble Programmer". In: *Communications of the ACM* 15.10 (1972), pp. 859–866

Since the hardware and software components are deeply intertwined, they can not be easily separated. Therefore, already established formal verification tools for software or hardware cannot be applied to integrated hardware/software systems in a direct manner. Several approaches tried to close the gap between integrated hardware/software design and scientifically approved verification tools, but all of them show weaknesses. In particular, all of the existing approaches share the problem of limited scalability. We give more details of related work in Chapter 2.

1.2 Research Objective and Criteria

Since deductive verification techniques have already turned out to be very powerful in reasoning about software with unbounded parameters, our research objective is to investigate if they can also be used for integrated hardware/software systems.
We require our approach to fulfill the following criteria: It should be semantics-preserving with respect to the interleaving of processes and the resulting program state. Furthermore, it must be scalable in number and size of system components. Interesting and representative problems should be identified and evaluated in appropriate case studies. Specifications like preconditions, postconditions, and invariants are mainly defined manually. We nevertheless aim at an extensible approach that allows for more automation.

1.3 Proposed Solution

Modeling embedded systems focuses more on the communication between components and less on the computation itself. It is established to design embedded systems actor-oriented and not object-oriented like common modern software [Lee02].
SystemC is a hardware support extension of the well known software language C++. It implements the already mentioned actor-oriented design and offers the possibility of software/hardware co-design [DG97]. SystemC is widely used in the area of embedded systems in both academic research and industrial contexts. Hence, we aim to study representative examples implemented with SystemC.
To verify the examples, we will explore deductive verification techniques. The VerCors Verifier is a tool for static verification of parallel programs developed by the Formal Methods and Tools Group, University of Twente [Ver21]. It uses Concurrent Separation Logic (CSL) [BO16] as its logical foundation. The advantages of VerCors are its language-independence, modularity, and the special handling of permissions. For example, we are able to abstract components by invariants like “the buffer size is always greater zero” and get the absence of race conditions without extra effort.
We will investigate a semantically correct and modular transformation from SystemC code into a valid input format for the VerCors Verification Tool. Furthermore, we will analyze the key concepts of SystemC like the simulation kernel with update-request scheme and event communication, and map them to semantically equivalent Prototypal Verification Language (PVL) code. PVL is VerCors’s own research language designed for reasoning about language-independent concurrent structures.

We mainly write the necessary specifications for VerCors manually, but we identify annotations which can be genereted automatically as well. If a PVL program passes the verification by VerCors, VerCors already guarantees the absence of data races, and memory safety. On top of that, we will formalize further safety and even partly liveness properties in VerCors's specification language, and verify them.

1.4 Organization of this Work

First, we give an overview about related work in Chapter 2 and explain why our approach differs from the existing concepts. We proceed with background information about Hardware/Software-Co-Design, SystemC and the VerCors Verifier in Chapter 3. In these preliminaries, the focus lays on the communication and concurrency modeling in SystemC, and the semantics and syntax of PVL and the Specification Language in VerCors. By understanding their key ideas, we are able to design a correct mapping between SystemC and annotated PVL code.

Our research and main part starts with Chapter 4, where we investigate a semantics-preserving transformation from SystemC to annotated PVL code. We contribute to the design of the transformation by providing (informal) transformation rules for an initial subset of SystemC constructs. Furthermore, we overcome the challenge of keeping SystemC's cooperative scheduling semantics by introducing a global lock mechanism. While the verification with VerCors produces extra effort by adding permissions to the input program, we identify a set of permissions and further specifications that could potentially be added automatically to ease the verification.

To emphasize the relevance of our work, we show how selected safety properties can be formalized with VerCors's specification language, and verify them. This is decribed in Chapter 5. We also compare our results with outcomes by another research approach to place them into scientific context. Both the transformation rules and the verified safety properties will be illustrated by simple, but representative examples.

Finally, we sum up the gained knowledge and achievements by our investigion of scalable verification of integrated hardware/software systems so far, and how future work can further enhance this. For this, we point out still existing challenges and outline promising ideas.

2
Related Work

In the past, various approaches to verify integrated hardware/software systems have been studied. Most of them comprise model checking techniques, i. e., properties are checked on an abstract model of the real system. They often suffer from the state space explosion probem, which reduces their applicability to real SystemC designs. To reduce the state space, some researcher have focused their work on the automatic partition of systems into separate hardware and software components, and to verify them separately. This reduces the model complexities, while the verification is still based on model checking. There also exist publications about the idea to split the formalization and verification processes. All of these approaches can not completely avoid the state space problem, thereby the verification of integrated hardware/software systems is still an open research question.
A more recent approach pursues deductive verification. It adds specifications to the system's code to generate proof obligations, which are discharged by interactive or automatic theorem provers, often involving Satisfiability Modulo Theories (SMT)[1] solvers. Still, this deductive approach is based on state-transition models, and therefore not completely scalable. However, there exist deductive verification techniques which are contract-based, but not yet investigated for the verification of integrated hardware/software systems.
In this chapter, we give an overview of the most recent approaches related to the verification of integrated hardware/software systems. We describe their key ideas, and outline advantages and weaknesses.

2.1 Model Checking Monolithic SystemC Designs

Herber, Fellmuth, and Glesner [HFG08] have developed **a mapping from SystemC designs to Uppaal timed automata (called STATE)**. The informally defined semantics of SystemC were completely preserved in the generated Uppaal models. This enables the

[1] SMT extends the Boolean Satisfiability (SAT) problem by adding reasoning about first-order theories like equalities, arithmetic or quantifiers.

verification of liveness, safety and timing properties by model checking timed automata. On the one hand, the translation was performed automatically and transformation time was negligible. On the other hand, it could not master the state space explosion problem. This has been demonstrated by their producer-consumer example whose verification time increased exponentially with the buffer size. However, the (restricted) use of STATE has been proven on several case studies, including an industrial design of an AMBA bus [HPG15].

Another approach by Karlsson, Eles, and Peng [KEP06] used **a design representation called Petri-net based Representation for Embedded Systems (PRES+)**. Maintaining the SystemC design's semantics, it has been translated into a PRES+ model. Afterwards, it is verified by Uppaal. Although the SystemC simulation kernel was explicitly modeled in PRES+, only limited support for SystemC features has been given. Furthermore, the experimental results show state space explosion again.

Besides timed automata and Petri-nets, **a transformation of SystemC designs into nondeterministic sequential C programs** has been explored. Herber and Hünnemeyer [HH14] developed this idea and used the BLAST model checker for verification of the C programs. BLAST offers counter-example guided abstraction refinement resulting in reduction of the verification time for interesting cases, like an Anti-Slip Regulation (ASR) and Anti-Lock Braking System (ABS). However, this approach still suffers from the state space explosion problem.

Efficiency improving techniques for model checking like symbolic execution, bounded model checking (BMC), or partial order reduction (POR) have also been applied, but they merely narrowed the state explosion and did not prevent it. Chou et al. [Cho+10] investigated a hybrid approach and **combined symbolic simulation, bounded model checking and invariant checking**, but they customized the SystemC simulation kernel. Hence, it covered only a part of the SystemC functionality. In addition, it only proved deadlock properties up to a certain bound but no other safety properties.

2.2 Automatic HW/SW Partitioning and Verification

Besides improving model checking efficiency, a lot of research has been done in the field of SystemC design partitioning. Due to the partition of integrated hardware/software systems into separated hardware and software components, the model size could be reduced before checking it and software- or hardware-specific verification techniques can be applied.

Kroening and Sharygina [KS05] **automatically divided up a uniform SystemC design into synchronous (hardware) and asynchronous (software) parts**. They syntactically distinguished combinational threads, clocked threads, and unrestricted threads from each other and modeled them as labeled Kripke structures. Still, the verification process is done by model checking and therefore suffers from state space explosion.

A different splitting has been proposed by Herber [Her14]. They **defined an intermediate representation for SystemC (SysCIR) and processed it by combining formal hardware verification, software verification and system verification (RESCUE)**. In addition, they

developed innovative slicing and abstraction engines for reducing the semantic state space. Herber and Liebrenz [HL20] enhanced the approach recently by **automatically partitioning SystemC designs into submodels and analyzing dependencies between them**. They could prove global properties of the original SystemC system by verifying them separately for each subsystem. Still the approach is not fully scalable in its application on general SystemC designs, especially if hardware and software are deeply intertwined.

2.3 Separation of Formalization and Verification Tool

Rather than dividing SystemC designs, Le et al. [Le+13] proposed to **separate the research for the formalization and verification processes**. The first part is always a transformation from SystemC to a formal model ("frontend"). It follows the development of a verification tool for this model ("backend"). They presented an Intermediate Verification Language (IVL) and checked it with their separately developed symbolic simulation tool SISSI. However, a disadvantage of the latter was its inability to detect loops.

To deal with this, Herdt et al. [Her+18] applied **symbolic subsumption checking for efficient detection of revisited symbolic states** in the model checking process and elaborated the state matching algorithm ESS. Still, the approach was not scalable. The support for SystemC features was limited by their IVL and no advanced techniques for alleviating path explosion were considered.

In 2021, Herdt, Grosse, and Drechsler [HGD21] improved their approach by several extensions and optimizations, such as SSR (State Subsumption Reduction) and CSS (Compiled Cymbolic Simulation). They could reduce the state space explosion, but not completely avoid it.

2.4 SMT Solving and Deductive Verification

Recently, promising results were achieved with verification of SystemC designs by using deductive verification techniques. Specifications have been added to a SystemC design and handed over to a theorem prover. All of the following approaches included SMT solvers as their backend reasoning tool.

Uclid is a SMT solver which offers finite precision bitvector arithmetics, so Jass and Herber [JH15] investigated a **bit-precise verification for SystemC by Uclid**. To face the state space explosion problem from model checking approaches, Uclid applied k-inductive invariant checking and used symbolic variables. Still, the approach suffered from many sequential parts of the concurrent simulation of SystemC designs. Since Uclid was designed for real parallelism (simultaneous execution), every sequential execution part increased the number of symbolic simulation steps necessary for the k-inductive verification and decreased the scalability.

Therefore, Schwan and Herber [SH20] **optimized the verification with Uclid in 2020 and tried to reduce inductive steps**. They focused on the symbolic simulation of cooperatively scheduled concurrent processes in Uclid and extended dominator trees to these. By analysing data, control, and inter-process dependencies of SystemC designs with dominator

trees, they could reduce the number of redundant states. Furthermore, they proposed a process parallelization based on a SystemC dependence graph. Thereby, Schwan and Herber achieved a significant reduction of verification time. Still, the approach with SystemC and Uclid suffered by the discrepancy of concurrency modeling in SystemC and Uclid.

2.5 Current State of SystemC Verifications

A year ago, Lin and Xie [LX20] investigated state-of-the-art SystemC verifications by discussing their methodologies, advantages, and limitations. To sum up their outcomes and the results of our literature check: All of the existing approaches either lack support for relevant SystemC features, or they do not show applicability to real industrial SystemC designs in consequence of bad scalability. Consequently, formal verification of SystemC is still an open and complex research topic.

The research studies with Uclid revealed that deductive verification of SystemC designs does not scale well if it is based on state-transition models. A completely new deductive approach for SystemC designs is contract-based verification. With the VerCors verification tool, preconditions, postconditions, and invariants of SystemC functions can be specified and checked. Parameters or complete functions can be abstracted and the verification is performed modularly; thereby our proposed solution scales well and meets our main research criteria.

3 Preliminaries

In this chapter, we give an introduction to SystemC and the VerCors Verifier. For both, we present background details to motivate our choice of modeling language and verification tool. Afterwards, we explain properties and advantages.

3.1 Hardware/Software Co-Design

A big advantage of SystemC is its enablement of early Hardware/Software Co-Design. If a developer wants to add a function to a system, it can be implemented in every level of the program. For example, an adder can be modeled either as a digital circuit or as a function in C++. The former is more efficient, but not flexible, the last is slower, but can be easily adapted to new requirements. This forces the developer to evaluate which layer is the best choice for the implementation of a new function—and this decision should be made as early as possible. The simultaneous development of hardware and software, which enables early design decisions, is called **Hardware-Software Co-Design**.

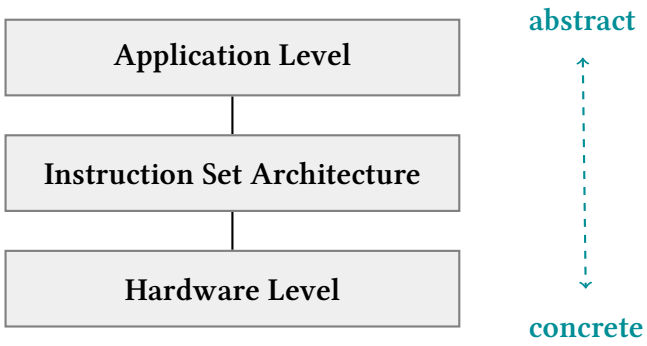


Figure 3.1: Technological Levels of Programming

We generalize the division into **different levels of programming** by following the definition of De Micheli and Gupta [DG97]. They are illustrated in Fig. 3.1. The application level is

the highest abstraction level and most flexible. Developers can use specified programming languages to add further functionality and usability to the system.

On the intermediate level, programming is done by using Instruction Set Architectures (ISAs). An ISA provides a behavior model of the system's hardware like microprocessors, microcontrollers, and Programmable Digital Signal Processors (DSPs). It forms the boundary between hardware and software. Normally, the ISA of embedded systems is not visible for the system's user, since it runs the embedded system. The lowest level of programming is the hardware level. Mostly, hardware is rather configurable than programmable. For example, there exist hard-wired Application-Specific Integrated Circuits (ASICs), which are extremely efficient, but not programmable at all.

During the last years, complexity of embedded systems has grown enormously. They have increased both in the number and size of their components. At the same time, their development still should meet as many economical criteria as possible: Less space requirement, less cost, less development time, less production time, higher performance, more flexibility/reusability. The choice of implementing functions in hardware or software is always a trade-off between these criteria. Hardware implementations are faster, use less power and space, but adjustments after their manufacturing are very costly. Software is more flexible and extensions can be developed also after the system's launch without further production costs, but it is less efficient than hardware.

The priority of above criteria depends on the system's purpose. There is no general optimal solution to design embedded systems, especially deeply intertwined hardware/software systems. Originally in this decision process, systems have already been partitioned into software and hardware before their design. They have been developed separately and merged together later. This is not possible for integrated hardware/software systems, since their components are closely linked together. A separation of them would cause many problems from increasing costs to unacceptable loss of performance. The solution is the concurrent design of hardware and software for meeting system-level objectives: **Hardware/Software Co-Design** [DG97]. The process of considering implementations either in hardware or software is called **Design Space Exploration**, because it happens during the design phase and not before. Design Space Exploration must take all complex criteria of embedded systems into account, but also the designated purpose of the new system.

SystemC offers not only Register-Transfer Level (RTL) modeling, but also more abstraction up to Transaction-Level Modeling (TLM). By this, it allows to model different parts of new implementations with differing granularities. Hence, Design Space Exploration can be performed very well with SystemC. Especially modeling on high level gives a very early understanding of the interaction between components. This results in **better system trade offs**, allows **earlier verification**, and achieves **overall productivity gains.**

3.2 SystemC

This section is based on two SystemC introductory books. Both give a comprehensive insight into SystemC, but with different priorities. Black and Donovan [BD04] focus more on the language statements itself and outline the principles fullfiled by them, whereas Grötker et al. [Grö+02] discuss the SystemC modeling functionality more generally.
First, SystemC is not an autonomous language. Instead, it is a class library of the well-established software language C++[1]. Significant extensions of C++ by SystemC are the support for simulation of hardware-oriented features like timing, communication and concurrency management, and hardware data types. Thus, SystemC enables Design Space Exploration as described in Section 3.1.
We give an overview of the SystemC language architecture in Table 3.1. The heart of SystemC builds an event-driven simulation kernel. It simulates concurrency by switching between processes until all of their executions are finished, or the simulation time ends. Processes can either wait for events to continue or notify that an event occured, so that other processes waiting for it can proceed. Actually, this type of concurrency simulation is not only used by SystemC, but also for Verilog [TM08], Very High Speed Integrated Circuit Hardware Description Language (VHDL) [RS20] and other Hardware Description Languages (HDLs). We describe more details of the concurrency simulation in Section 3.2.3.

IEEE SystemC Standard

TLM	AMS	SCV	User Libraries
Predefined Channels Clock, Mutex, Semaphore, FIFO, Signal			
Core Language			**Data Types**
Modules Threads Methods	Channels Interfaces Ports	Events Notifications Sensitivity	Logic Types Bits and bit-vectors Arbitrary-precision integers Fixed-point numbers
Event-driven Simulation Kernel			
Programming Language C++			

Table 3.1: SystemC Architecture.

Processes in SystemC are encapsulated into classes which are called **Modules**. Depending on the process type, either a **Method** or a **Thread** holds the process execution statements. We furtherly explain this in Section 3.2.1.
To abstract the communication between modules and their environment, **Ports** are introduced by SystemC. The interaction between processes is abstracted by **Channels**. They can be modeled very detailed or on a highly abstracted level. The processes do not notice the level of abstraction. They only connect to the channels' **Interfaces**, which define the functionalities offered by the channel.

[1] To be precise, SystemC is based on C++ and the C++ Standard Template Library (STL).

To model the concurrent execution of processes, they can wait for **Events** and suspend their execution. Also processes can notify that an event has occured. Not all processes are observing all events. Their relations are built by their **static or dynamic sensitivity** to events. We discuss the full communication concept in Section 3.2.2.
The mentioned simulation kernel, structural components and communication management form the SystemC Core Language. Together with newly introducted data types for hardware modeling and several predefined channel implementations, they form the **IEEE SystemC Standard** [IEEE11], indicated by the teal highlighting in Table 3.1. The first IEEE SystemC Standard has been published in 2005 and revisioned in 2011.
In addition to the SystemC core language components, several **methodology- and technology-specific libraries** have been developed [Acc21c], such as:

- The library for Transaction-Level Modeling (TLM) and virtual prototyping,
- the Analog/Mixed-Signal (AMS) library for system-level design and modeling of AMS systems, and
- the SystemC Verification Library (SCV) providing APIs usable as a basis to verification activities with SystemC, e. g., generation of values under constraints.

These libraries are only partly or not yet IEEE-standardized, but get continously upgraded by the Accellera Systems Initiative [Acc21a] and used in practice. There also exist further user libraries developed by the SystemC community groups.
In this work, our focus is on the possiblities to combine the SystemC core language with the VerCors Verifier functionalities. To get a deeper understanding of the SystemC core, we investigate the details in the following subsections.

3.2.1 Structural Components

For integrated hardware/software systems, it is rather interesting how the system components interact—according to a model of computation—instead of how they precisely compute. The computation or algorithms themself are often abstracted. Lee [Lee02] calls this **actor-oriented** design instead of object-oriented design.
The actors in SystemC are described by modules, which are C++ classes declared by the macro `SC_MODULE`. Technically, the macro derives the class from the SystemC library class `sc_module`. We give a detailed code example of a module in Listing 3.1.
A module holds the elements explained in the following paragraphs. All of them are optional except for the constructor. Like in standard C++, it is possible to divide declaration and implementation of the elements into a .h file and a .cpp file. For simplicity, we will not seperate them in our code syntax examples.

Ports

Modules communicate via ports with their environment. For example, values measured by external sensors can be modeled with a port. They also facilitate to connect one module to another (cf. Section 3.2.2). The SystemC library class `sc_port<?>` offers the port functionality.

```
1  #include <systemc.h>
2
3  SC_MODULE(moduleName)  {
4
5    // Ports
6    sc_port<if> portOfInterface;
7    sc_port<sc_clock> portOfClock;
8
9    // Member data instances
10   int numberOfPorts;
11
12   // Process functions
13   void methodProcess(void)  {
14     // Immediate computations
15   }
16
17   void threadProcess(void)  {
18     while(true) {
19       // Interruptable process implementation
20     }
21   }
22
23   // Constructor
24   SC_CTOR(moduleName) {
25     // Process declaration, sensitivity, subdesign tasks
26     SC_METHOD(methodProcess);
27     SC_THREAD(threadProcess);
28     sensitive << portOfClock;
29   }
30
31   // Member submodule instances
32 };
```

Listing 3.1: SystemC Module Example.

The question mark is a wildcard for any C++ or SystemC class. In our example, the first type in line 6 is an interface. The second one in line 7 is a clock, which is very useful for hardware simulation of clock cycles.

Data and Channels Instances

Like any other C++ class, a module can have class or member variables to keep data of the module state. An example of an integer member variable is presented in line 10. Furthermore, if the module contains more than one process, internal channels for the communication between these processes could be declared and instantiated.

Processes

The core element of a module is a SystemC process, implemented by a member or class function. They are restricted to have no return value and do not take any parameters, since the SystemC scheduler is the only caller of these functions. To register a function as

a process with the simulation kernel, the macros `SC_THREAD` or `SC_METHOD` are used. Threads and methods have different meanings in SystemC.
A **Method Process** is a function in which no simulation time passes. So it is useful for short, simple computations and can be repeatedly called by the SystemC scheduler (cf. Section 3.2.3).
A **Thread Process** is both a thread of execution and a modeling of independently-timed circuits, depending on a software or hardware design view. It is invoked only once by the scheduler, but can suspend its execution and continue at another time. This is why nearly every SystemC thread implements a `while(true)` loop. The process ends when the thread exits or returns.
All functions not declared as processes in the constructor are normal C++ member/class functions. They can be called by method/thread processes like by any other function.

Constructor and Destructor

The constructor is declared by the macro `SC_CTOR` and the module name. Technically, the constructor registers processes with the SystemC kernel and establishes the static sensitivity to events or channels (cf. Section 3.2.3). The static sensitivity is declared by `sensitive << event/channel` and always holds for the most recently declared process. In our example, this is the thread process, but *not* the method process.
The constructor also initializes subdesigns and their connection or can hold further user-defined tasks. SystemC does not define a special destructor, but the standard C++ destructor can be used to free memory occupied by objects.

Further Submodules

The partition of a design into modules can also go deeper by adding submodules as members to a parent module. Since we will not use them in our case studies in this work, we will not explain them in detail and refer to the SystemC introduction by Black and Donovan [BD04].

3.2.2 Communication Modeling

In Section 3.2.1, we already mentioned that modules can communicate via ports with each other. Now we describe the details of this intercommunication. To give a better overview, Fig. 3.2 shows a general SystemC design holding all basic communication components.
In the context of hardware modeling, SystemC modules can be compared to blocks and ports to pins. Hardware designs use wires or signals to connect the blocks via their pins. The SystemC communication components are developed by keeping this architecture in mind.
Modules are connected by **Channels**, which offer a broad range of complexity, from a simple First-In–First-Out Queue (FIFO) up to an in-depth complex design like the Advanced Microcontroller Bus Architecture (AMBA). To keep channels maximally interchangeable,

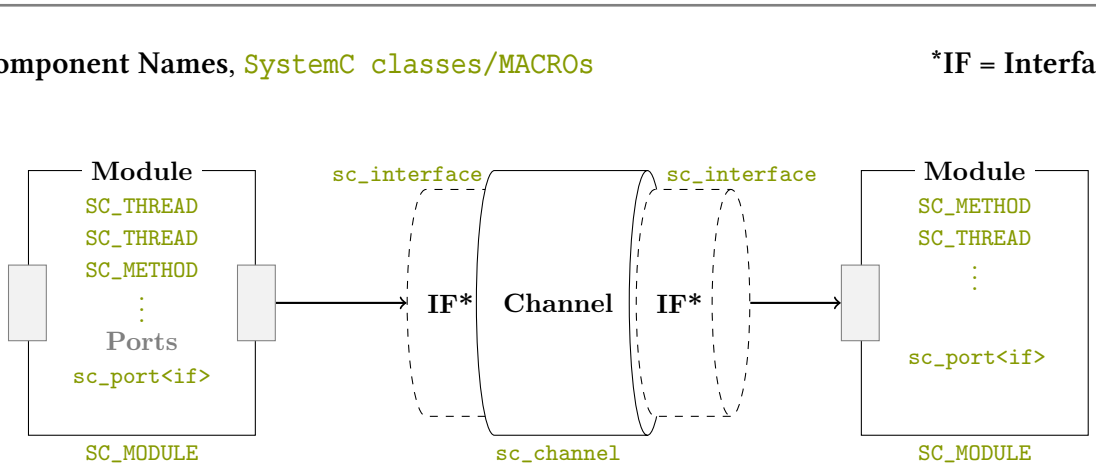


Figure 3.2: Overview of SystemC Communication Components.

they are abstracted by **Interfaces**. Like already approved by standard C++, the declaration and implementation of functions can be separated into a header file and a cpp file. SystemC transfers this approach and let the interfaces declare which functions are available, but the channels implement the real functionality. By this, different implementation levels of channels can be easily interchanged, as long as they implement the same interface.

Interfaces

To define a new SystemC interface, the SystemC class `sc_interface` is used as a base class. In Listing 3.2 we give an example of a simple read and write interface.

```
1 #include <systemc.h>
2
3 class readWriteInt_if : virtual public sc_interface {
4   public:
5     virtual int read(void) = 0;
6     virtual void write(int data) = 0;
7 };
```

Listing 3.2: SystemC Interface Example.

Our interface named `readWriteInt_if` defines two functions. The first one is a read function, which takes no arguments, but always returns an integer value. The second one is a write function, which takes an integer data, but returns nothing. They keyword `virtual` is standard C++ and denotes, that the class or function might be overriden by a derived class and its functions. The derived class will be the channel. The syntax `= 0` declares the functions `read` and `write` as *pure virtual functions*, that means our interface class is an abstract class from which no objects can be instantiated.
Instead, an interface is connected to a module via a port, as we showed in Listing 3.1, line 6. The only (correct) access to a channel is by using the functions offered by the interface

of the port. Hence, every module with a port instance of `sc_port<readWriteInt_if>` can call `read` and `write` from Listing 3.2. If the access to a channel should be more restrictive—for example, when a module shall only have read access—another interface must be used. In Listing 3.3 and Listing 3.4 we split the read and write functionalities into two different interfaces.

```
#include <systemc.h>
class readInt_if : virtual public sc_interface {
  public:
    virtual int read(void) = 0;
};
```

Listing 3.3: SystemC Read Interface Example.

```
#include <systemc.h>
class writeInt_if : virtual public sc_interface {
  public:
    virtual void write(int v) = 0;
};
```

Listing 3.4: SystemC Write Interface Example.

Now read and write access to the channel are separated, but still decoupled from the channel. With this architecture, we can apply our example interface source code to our design from Fig. 3.2. It is presented in Fig. 3.3. The interface on the left only offers write access, while the interface on the right provides read access. Still the channel has to implement the communication details in the functions `read` and `write`. We investigate channels in the next section.

Component Names, `SystemC classes/MACROs`, `Functions`

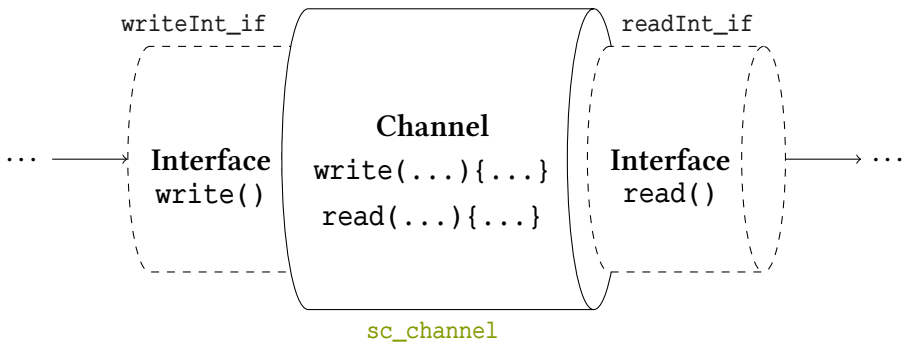


Figure 3.3: Read/Write Interfaces connected to Channel in SystemC.

Channels

To implement a channel in SystemC, a class must be derived from the library class `sc_channel`. Note that in contrast to many other object-oriented languages (e g., Java), C++ realizes the concept of multiple inheritance. Therefore, the channel must be derived from at least one interface to offer access from the outside. It has to implement all pure virtual functions of these interfaces. In our syntax example in Listing 3.5, both the interfaces `readInt_if` and `readWriteInt_if` are realized. A channel has few limitations on its complexity by the SystemC base class. Like any normal C++ class, it could have member/class data, constructor, and member/class functions.

```
1  #include <systemc.h>
2
3  class channelName : public sc_channel,
4      public readInt_if, public writeInt_if {
5
6    // Internal data
7    int value;
8
9    public:
10     // Constructor initializes value with 0.
11     channelName() : value(0);
12
13     int read(void) {
14       return value; // Implementation of readInt_if
15     }
16
17     void write(int v) {
18       value = v;  // Implementation of writeInt_if
19     }
20 };
```

Listing 3.5: SystemC Channel Example.

The IEEE standard of SystemC has several predefined channels [IEEE11, chapter 6]. We mentioned some of them in Table 3.1: `sc_clock`, `sc_mutex`, `sc_semaphore`, `sc_fifo` and `sc_signal`. All of these built-in channels implement corresponding predefined interfaces, for example, `sc_fifo` implements `sc_fifo_in_if`.

The predefined channels are called **primitive**, because they can not contain any SystemC structures like modules, processes, ports or other channels. Primitive channels are derived from the class `sc_prim_channel` and have two main advantages: They are very fast and they realize the **Request-Update Scheme**. This is related to the SystemC simulation kernel, so we handle it later in Section 3.2.3.

All channels which are not primitive, but simply derived from `sc_channel`, are classified as **hierarchical channels**. A possible point of view is that hierarchical channels are modules implementing interfaces. They are useful in the modeling of complex communication architectures like the AMBA. By this SystemC is more convenient for hardware modeling than standard HDLs.

3.2.3 Concurrency Simulation

So far we explained the basic structural components of SystemC and the boundaries of their communication. Now we focus on how these communication principles are used in the simulation of concurrency.

Simulation Starting Point

Every system needs a starting point. In general, software programs have an entry point, e. g., a main function in C++. In SystemC, the corresponding function is named `sc_main`; we give an example in Listing 3.6.

```
1  #include <systemc.h>
2
3  int sc_main(int argc, char* argv[])  {
4
5    // Elaboration Phase
6    channelName channel_inst("Example Channel");
7    moduleName  module_inst("Example Module");
8    module_inst.portOfInterface(channel_inst);
9
10   // Simulation
11   sc_start();
12
13   // Post Processing
14
15   // Simulation Status
16   return EXIT_CODE;
17 }
```

Listing 3.6: SystemC Starting Point of Simulation Example (`sc_main`).

The SystemC main function takes two arguments like the standard C++ main function. The first one `argc` is the number of arguments given by the command line calling the `sc_main`. The second `argv[]` is (the pointer to) an array which holds the full command line call, so the array element at index 0 is the program name.
Inside of `sc_main`, three phases can be distinguished:

1. **Elaboration Phase:** In this phase, all modules, channels, and other structures are initialized and connected. An important part is the **binding of ports to channels**. First, the module and channels are instantiated by calling their constructors (lines 6f). Then, the channel is passed as an argument to the port (line 8). The interface declared at the port in the module defines how the access to the channel is designed.
2. **Simulation:** By calling the function `sc_start()`, the simulation execution is started in the simulation kernel. We explain the simulation in detail in Section 3.2.3.
3. **Post Processing:** Optionally, results or other data of the simulation could be reported in the post processing. Also an exit code states the success of the simulation. If no problems occur, it is zero.

The most interesting phase is the simulation. If no specialized concurrency mechanisms are used in modules and channels, the SystemC simulator will call processes one by one, fully sequential. This is not the aim of SystemC, since hardware does not behave sequential at all. So to simulate concurrency, SystemC processes can suspend itself and continue their execution later. They need to synchronize with each other during the simulation. This is modeled by **Events**.

Events, Notifications, and Sensitivity

Events mark a specific state or the change of it in the SystemC mode, for example that no space is left in a FIFO. An event occurs at a specific point in time, but has no further arguments or a duration. To add a new event to a module, an object of the class `sc_event` must be declared, see Listing 3.7 (line 6).

```
1  #include <systemc.h>
2
3  SC_MODULE(moduleWithEvent)  {
4
5    // Events
6    sc_event eventName;
7
8    void triggerThread(void)  {
9      while(true) {
10       eventName.notify(); // eventName has happened.
11       wait(); // Waiting for a never occuring event.
12     }
13   }
14
15   void observerThread(void) {
16     while(true) {
17       wait();// Waiting for eventName by static sensitivity.
18     }
19   }
20
21   // Constructor
22   SC_CTOR(moduleName) {
23     // Process and sensitivity declaration
24     SC_THREAD(triggerThread);
25
26     SC_THREAD(observerThread);
27     sensitive << eventName;
28   }
29 };
```

Listing 3.7: SystemC Event and Static Sensitivity Example.

An event itself does not do anything during the simulation, but its occurence can trigger reactions of processes or channels. To state that an event happened, a function `notify()` can be called on the event. Not all processes are aware of all happening events. Instead, the SystemC simulation kernel notices all event occurences. If a process should be aware

of an event occurence, it has to be **sensitive to** and **waiting** for it.
SystemC distinguishes between **Static Sensitivity** and **Dynamic Sensitivity**. An example of the former is given in Listing 3.7, line 27. Static sensitivity means that the sensitivity is established during the elaboration phase in which the module constructor is called. The `observerThread` is declared as sensitivite to `eventName`. If the thread is waiting, as in line 17, it will continue its execution after `eventName` is triggered. The occurence of `eventName` is stated in line 10 by the `triggerThread`.
Note that the `triggerThread` will suspend itself in line 11 and never continue, since no further event notifications occur. The already stated notification of `eventName` does not apply, since the thread has not been waiting for it before the notification happened. A further remarkable detail is that a thread can be sensitivite to more than one event. So the `wait()` statement could also be fulfilled by an event other than the `eventName`, if the static sensitivity of the `observerThread` is declared to the other event in the constructor as well.
Dynamic Sensitivity is established during the simulation of the SystemC model. Instead of declaring `observerThread` as sensitivite to `eventName` in the module constructor, we could drop this statement and declare the sensitivity by passing `eventName` as an argument to the `wait()` statement. We give an example in Listing 3.8.

```
1  #include <systemc.h>
2
3  SC_MODULE(moduleWithEvent)  {
4
5    // Events
6    sc_event eventName;
7
8    void triggerThread(void)  {
9      while(true) {
10       eventName.notify(); // eventName has happened.
11       wait(); // Waiting for a never occuring event.
12     }
13   }
14
15   void observerThread(void) {
16     while(true) {
17       wait(eventName);  // Waiting by dynamic sensitivity.
18     }
19   }
20
21   // Constructor
22   SC_CTOR(moduleName) {
23     // Process declaration without sensitivity lists
24     SC_THREAD(triggerThread);
25     SC_THREAD(observerThread);
26   }
27 };
```

Listing 3.8: SystemC Event and Dynamic Sensitivity Example.

Dynamic sensitivity provides more flexibility. For example, we could add a conditional statement around the `wait(eventName)` statement to restrict the waiting to special cases. This is not possible with the static sensitivity list of the process.
In Listing 3.7 and Listing 3.8, events are members of modules which hold more than one process. However, we described in Section 3.2.2 that modules do not communicate directly with each other, but instead via port-bounded channels. Hence, it is only natural that channels use events as well, like presented in our example in Listing 3.9.

```
1  #include <systemc.h>
2
3  class channelName : public sc_channel,
4      public readInt_if, public writeInt_if {
5
6    // Internal data
7    int value;
8    sc_event writeEvent;
9
10   public:
11   // Constructor initializes value with 0.
12   channelName() : value(0);
13
14   int read(void) {
15     wait(writeEvent); // waits for write action.
16     return value;
17   }
18
19   void write(int v) {
20     writeEvent.notfiy();  // new value is written.
21     value = v;
22   }
23 };
```

Listing 3.9: SystemC Channel with Events Example.

We added an event `writeEvent`, which marks that a new value has been written into the channel. If a process calls the channel's `read` function, the process will suspend its execution by `wait(writeEvent)` until the `writeEvent` has happened. Note that another process must now call the `write` function, otherwise the first process gets stuck in its `read` call. This is a first hint which kind of bugs can be easily produced in SystemC code, e. g. **Deadlocks**.
Instead of listening to events, a process can also be sensitive to a channel. That means that any event occuring in this channel will trigger the process's `wait()` statements. In line 28 of Listing 3.1 (p. 13) of a module, we declare sensitivity to the built-in SystemC clock channel. Every time an event occurs in the clock channel, a `wait()` statement within the `threadProcess()` will be triggered.
Above, we wrote that processes can suspend their execution and continue later, but we did not yet clarify what *later* means. The *timing* of processes is related to the simulation kernel and therefore to the SystemC scheduler. This is the topic of our next subsection.

Simulation Kernel and Scheduling

The elaboration phase and post processing have already been described in Section 3.2.3. The code in between is the simulation starting with the call of `sc_start`.
First, we clarify the terms simulation, simulation kernel, and scheduler. The **simulation kernel** is part of the SystemC language architecture, which holds all components necessary for the simulation. Technically, it is single threaded and SystemC has no real concurrent executions. Instead, the **simulation** of concurrent actions is event-driven and orchestrated by the **scheduler**. The scheduler controls the simulation time, handles events notifications, and schedules the execution order of the processes. It is the most important component of the simulation kernel.
The simulation kernel holds different pools of processes depending on their state. They are either **ready**, **running**, **waiting**, or **finished**. We explain how they fit to the different phases of the concurrency simulation. For this, we start with the illustration in Fig. 3.4, which shows the first part of the simulation.

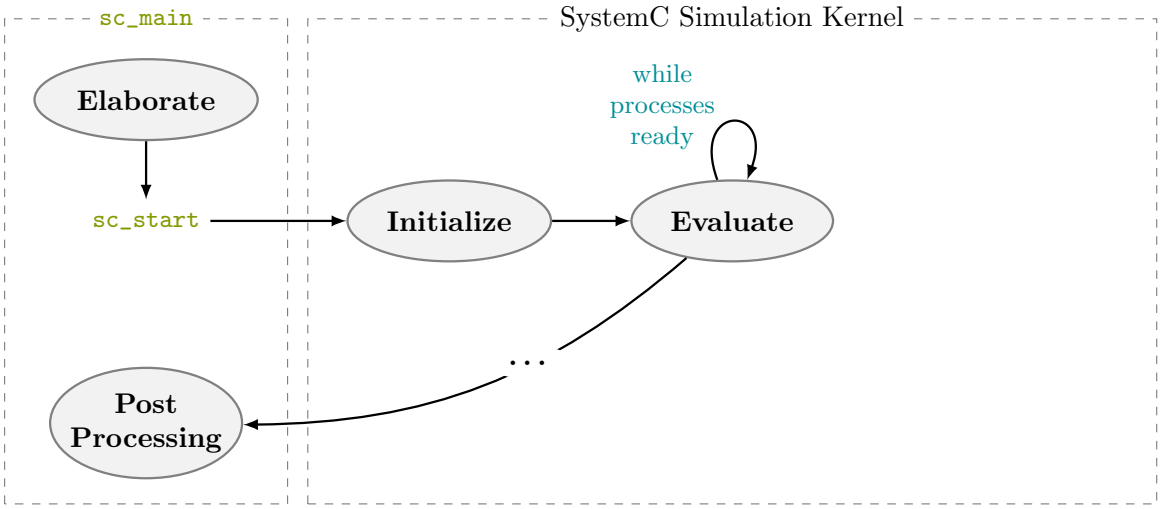


Figure 3.4: SystemC Simulation Kernel and Scheduling (without Concurrency).

In the **Initalization Phase**, all processes are placed in the ready pool by the scheduler. Now the scheduler invokes one process, so its state changes to *running*. This process executes until it encounters a `wait` statement, so it decides when to interrupt its execution. The process would also stop if it finishes and exits, but since embedded systems are normally designed to *never stop running*, this is a rare exception. The scheduler *can not* interrupt the process, so the scheduling algorithm is designed as **non-preemptive**. This sort of concurrency simulation is also called **cooperative multi-tasking**.
When a process encounters a `wait` statement, it is moved to the waiting pool by the scheduler. Then the next ready process gets invoked. This is repeated as long as ready processes are left. We call the repeated invocation of ready processes the **Evaluation Phase**. The interesting point is now how the processes are put back into the ready pool. For this, we have to consider different types of `wait` statements in SystemC. We explained

the event-related wait statements in Section 3.2.3, but now we must also take *timed* wait statements into account.
The following syntaxes of wait statements are an excerpt. There exist even more `wait` statements in the SystemC Language Reference Manual (LRM) [IEEE11]. In this work, we only use the following statements:

- `wait()`: If no argument is given to the `wait` call, the process waits for an event of the static sensitivity list to occur. This is kept in mind by the scheduler. If an event occurence is notified (by another running process), the scheduler switches the waiting process from waiting to ready pool.
- `wait(event)`: If the passed event occurs and its notification is made by the running process, the waiting process is placed back in the ready pool.
- `wait(double,sc_time_unit)`: After a specified time, the process is switched from waiting to ready pool. Either the time unit in addition to the amount of time must be given, or a time constant—like zero—is passed.
- `wait(double,sc_time_unit, event)`: This call behaves like `wait(event)`, but has a specific timeout. If the notification about the event occurence is made within the timeout, the scheduler places the waiting process back in the ready pool, otherwise not.

Up until now, we described how processes get back to the ready pool. Currently, concurrency is only established by the interleaving of processes. Still, in reality, hardware blocks are not operating sequentially. Also note that the execution order of ready processes is not defined in SystemC (non-determinism), but crucial in reality to get correct results. To achieve a realistic model of concurreny, SystemC uses the concept of **Delta Cycles**, similar to Verilog or VHDL.

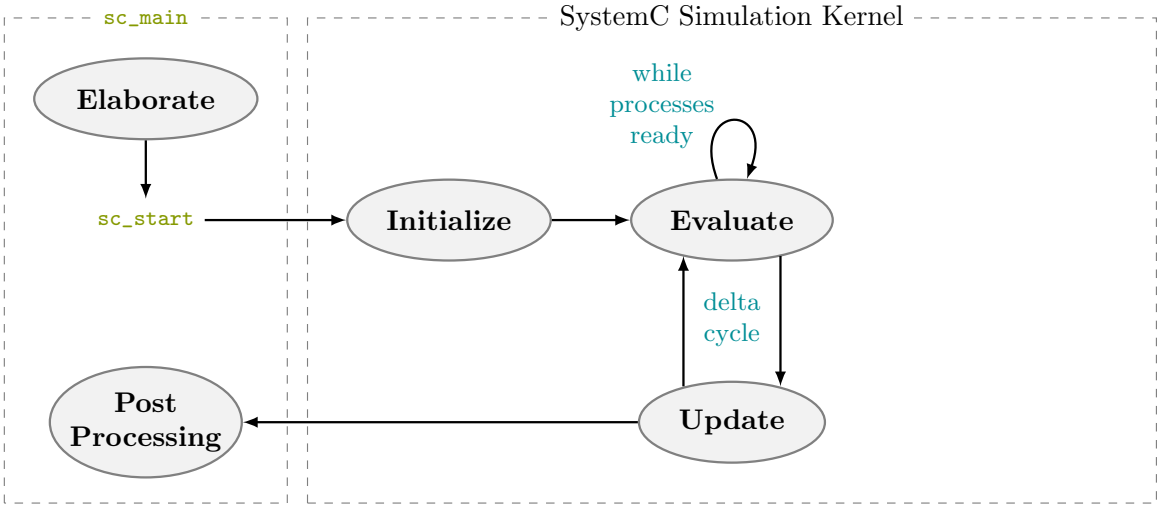


Figure 3.5: SystemC Simulation Kernel and Scheduling (without Time Advancement).

Newly computed values are not directly set, but only *evaluated* in the evaluation phase. Afterwards, when no process is left in the ready pool, the value outputs get updated in

the separated **Update Phase**, which is illustrated in Fig. 3.5. This is also known as the **Request-Update Scheme**, which is implemented by all primitive SystemC channels. It is also possible to implement channels, whose values are directly updated in the evaluation phase. However, this should be done very cautiously due to the mentioned reasons.
One evaluation phase followed by an update phase is defined as a **delta cycle**. During a delta cycle, no simulation time passes and behavior of real parallelism is simulated. There can be multiple delta cycles before the simulation time advances. This is due to the type of event notifications which may be timed or not:

- `event.notify()`: This `notify` statement is called an **Immediate Notification**, since it lets the scheduler place all processes—waiting for the given event—in the ready pool *within the current delta cycle.*
- `event.notify(0)`: If the constant `SC_ZERO_TIME` or shortly `0` is passed as an argument, a **Delta Notification** of the event is made. This means, that the switch of the process from waiting to ready pool is made *within the next delta cycle.*
- `event.notify(double,sc_time_unit)`: This event notification will be not noticed in the current time step, but after the given amount of time. It is called a **Timed Notification**, because it is considered by the scheduler *after a time advancement.* The time argument is either passed as the amount of time and the time unit, or as a time constant that is not zero.

As we previously mentioned, no time passes within or between delta cycles. All actions within or between delta cycles are considered as concurrent operations. As long as ready processes exist after the update phase, a new delta cycle is started. Only if the ready pool is empty, then the scheduler makes a **Time Advancement**. The scheduler looks for the nearest point of time causing a waiting process to be switched to the ready process. By this, unneccesary time steps are avoided and the SystemC simulation efficiency improved.

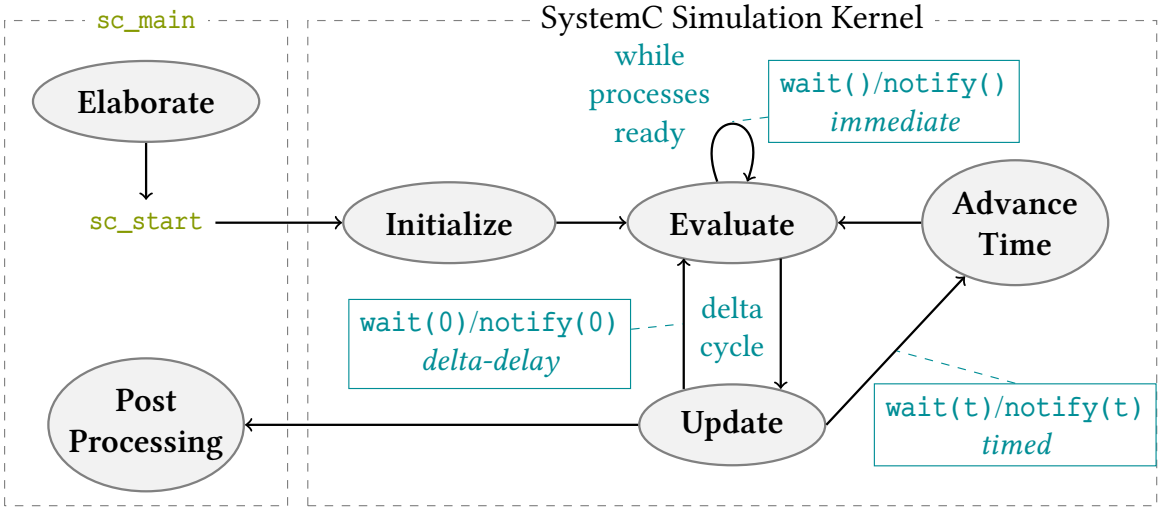


Figure 3.6: SystemC Simulation Kernel and Scheduling.

If no further ready processes, update requests, timeout wait statements, delta notifications, or timed notifications and timeouts are left—so no more process activity is possible—the simulation ends and a cleanup in the post processing is done. It is also possible to call `sc_start(double, sc_time_unit)` with an amount of time as an argument to restrict the simulation time in advance.
We summed up the simulation phases and corresponding `wait`/`notify` statements in Fig. 3.6, which is also based on the work by Herber [Her10, fig. 2.11]. Note that the described simulation semantics primarily hold for thread processes in SystemC. Since method processes can not call any `wait` statements, they execute without interruptions and return to the scheduler afterwards.
Another often misleading assumption is that all timed notifications of an event are kept by the scheduler. This is not the case, since **an event is allowed to have only one outstanding timed notification**. A notification of time t_1 will overwrite the notification of time t_2 if t_1 is smaller than t_2.

3.2.4 Main Advantages of SystemC

In the previous sections, we described the SystemC language components and simulation semantics. We now explain why SystemC is very suitable for our research purpose. As mentioned, SystemC behaves similar to Verilog, VHDL, or other HDLs in key functionalities. This covers for example the support of hardware data types or the type of concurrency modeling. It is also very convenient that SystemC has an open IEEE standard. This facilitates easy plug-in extensions. As a result, many famous hardware designers have contributed to SystemC for more than 20 years up to now. At the same time, real-time operating systems are not really standardized in practice. They are specific to their scheduling, timing, eventing, and more. Despite this fact, the well understandable simulation semantics and nice tooling emphasize SystemC as a good general representative for all concurrent system models.
To sum up all outcomes of this chapter, there are three main reasons why we have chosen SystemC as our modeling language:

1. SystemC extends the well-established software language C++ by support for the simulation of hardware elements. By this, it is a **very powerful language for modelling both hardware and software** at the same time.
2. SystemC evolved by the need for early co-design of hardware and software, especially if they are deeply intertwined. It **enables system design trade-offs on various levels of abstraction**: Executable Specifications, (Un)timed Functional Models, TLM, Hardware Models of different granularities down to RTL models. Therefore, implementations in either hardware or software can be discussed early, and development time and cost get reduced significantly.
3. In history, SystemC has been the attempt to create a more unified industry standard. There exists a wide bunch of real-time operating systems in practice which are specific to their concurrency, real-time modeling, eventing, and more. While SystemC

can not cover all of them, it is **easy to extend tools for SystemC to other runtime environments**.

We now continue to introduce the tool for verifying our SystemC designs.

3.3 VerCors Verifier

The VerCors Verifier[2] is a tool for **static verification of concurrent and distributed programs** developed by the Formal Methods and Tools Group, University of Twente [Ver21]. Static verification means that no execution of the program happens; instead, program statements and their *annotations* are analyzed. This approach lets VerCors scale nicely with increasing size of program parameters. Therefore, it fits very well to our research objective (cf. Section 1.2).
VerCors is designed to be **language-independent**. It enables direct verification of annotated Java, C, OpenMP and OpenCL programs, but can be extended to new frontend languages easily. In addition, VerCors has its own Prototypal Verification Language (PVL), which realizes a very general concept of concurrency and parallelism. In Listing 3.10, we give a simple example written in PVL.

```
1  class Counter {
2    int val;
3
4    requires Perm(val, 1);
5    ensures Perm(val, 1);
6    ensures val == \old(val) + n + 25;
7    void incr(int n) {
8      val = val + 42;
9      val = val - 17;
10     val = val + n;
11   }
12 }
```

Listing 3.10: PVL Simple Counter Example.

The presented counter has a variable `val`, whose value is increased by the function `incr`. The specification `Perm(val,1)` denotes write permissions of the variable `val`. The function requires its caller to have these permissions (*precondition*) and to *give them back* afterwards. Under this assumption, the function ensures that the value of `val` after execution of `incr` is equal to the previous value of `val` increased by the argument `n` and 25 (*postcondition*).
To verify the assertion of the new value of `val`, we call the PVL file with the option `--Silver=silicon`. It sets the used verification tool to Silicon, which is part of the Verification Infrastructure for Permission-based Reasoning (Viper) [MSS16]. We give further information about VerCors's logical background and architecture in Section 3.3.1. For our example of a counter, the verification result is `Pass`, as presented in Listing 3.11. In general, the

[2]The name origin is inspired by a mountain massif called *vercors*, which is displayed in the website's header image.

other verification result is `Fail`, but in this case VerCors would provide an error message whether the assertion did not hold or what other problems occured.

```
$ vercors --Silver=silicon counter.pvl
Success!
The final verdict is Pass
```

Listing 3.11: VerCors Verification Call and Result

In the following sections, we describe VerCors's theoretical foundation and architecture, the general syntax of PVL, and the Specification Language for annotations. After understanding the basics of both PVL and the Specification Language, we provide a more detailed section about concurrency modeling and verification with VerCors. We sum up our explanations by outlining the main advantages of VerCors. For all sections, one of our main literature references is the VerCors website [Ver21]. It is managed by VerCors's creator, the Formal Methods and Tools Group, University of Twente, and provides multiple helpful pages to get introduced to VerCors. A short website guide can be found in Table A.1.
We also refer to selected publications about VerCors and cite them in the corresponding subsection. Since VerCors is an open source project, its full source code is available and hosted on GitHub [Ver21g].

3.3.1 Logical Foundation and Architecture

In this section we give a short introduction to VerCors's backend architecture and the verification process. Afterwards, we focus on the logical foundation. That means we will investigate the mathematical, logical concepts on which VerCors is built.

Technical Backend Architecture

VerCors is not verifying properties standalone, but rather provides a convenient and well-structured frontend of a powerful backend. The tool architecture is illustrated in Fig. 3.7.

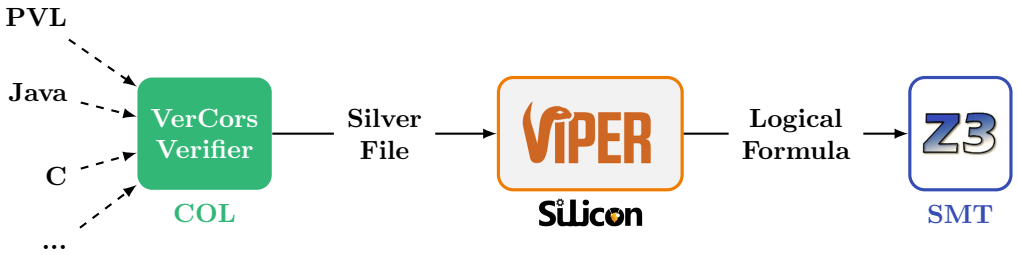


Figure 3.7: VerCors Verifier Tool Architecture (with Silicon)

VerCors takes a program written in PVL, Java, C, or one of the other supported input languages and translates them into its intermediate data structure, an Abstract Syntax Tree (AST). This repesentation is expressed in the Common Object Language (COL)

[Ami+12]. VerCors parses the AST multiple times to perform type checks and other improvements. Afterwards, it encodes the AST into **Silver**, the intermediate verification language of **Viper**.
The Viper tool suite is developed and maintained by the Programming Methodology Group, ETH Zürich [Vip21]. It offers a verification-condition-generation-based verifier named *Carbon* and a symbolic-execution-based verifier named *Silicon* [Sch16]. Previous releases of VerCors only support Silicon, but the most current version (1.4.0) also enables to use Carbon. Up until now, there exist no scientific publications comparing Silicon and Carbon, but Kassios, Müller, and Schwerhoff investigated the efficiency of condition generation versus symbolic execution exemplified by Chalice and Syxc [KMS12]. They concluded, that symbolic execution generally is more efficient, so we see Silicon as a good choice for our work. Note that the architecture of Viper slightly differs for Carbon, so Fig. 3.7 shows the tool architecture when using Silicon.
The Silicon verifier translates assertions written in Silver into logical formulas, precisely in SMT-LIB format [BST+10]. Then, it passes them to the **Z3 SMT Solver**, a theorem prover from Microsoft Research [DB08]. When Z3 solves a formula, it decides whether the formula is satisifiable or not. The result is given back to Viper and from Viper to VerCors, which presents a human-readable feedback related to the input program. Note that the feedback given back by Silicon is a bit different than the one by Carbon. Hence, for cases which do not pass, it could be helpful to switch between Silicon and Carbon to yield more feedback. To prove a formula, it is also useful to check whether the negation is unsatisfiable.

Logical Foundation

We mentioned that VerCors realizes a frontend for well-established backend architectures. Now we describe on which logical concepts VerCors itself is built. For this, we refer to the work by Amighi, Blom, and Huisman [ABH16].
VerCors's logical foundation is a combination of Implicit Dynamic Frames (IDF) [SJP09] and Concurrent Separation Logic (CSL) [BO16]. IDF is a program logic based on Hoare Logic and extends it by reasoning about access permissions to heap locations. VerCors's `inhale` and `exhale` statements are complementing IDF for language constructs that are implemented by VerCors, but not part of IDF.
CSL is a program logic for concurrency and provides **fractional permissions**. A thread can only access shared memory from the heap if it owns *sufficient* permissions. We already presented an example in Listing 3.10, where `Perm(val, 1)` denotes write permissions. Only a value of `1` enables write permissions. Any other fraction greater than zero and less than one denotes only read permissions. A value of `0` marks that the thread has no permissions at all. We sum up the relation between permissions and their numerical range in Table 3.2. The **permission fractions of all threads to a certain location must always sum up to 1**. As a result, write permissions are always *exclusive*. Permission handling is a notable

No Permission	Read Permission	Write Permission
0	$0 < v < 1$	1

Table 3.2: VerCors Permissions for Threads.

specification overhead requested by VerCors, but **it guarantees data race freedom and memory safety without further effort**. So the use of VerCors for sequential program verification may be exaggerated, but for concurrent programs it is well suited.
Besides its logical base, VerCors differentiates between verification layers, depending on the properties which should be verified. The first layer enables the **check for data race freedom** and **memory safety** in shared-memory concurrent programs. However, the deduced invariants about memory resources are proven **thread-locally**. If this is not sufficient for the considered properties, another layer is applied, which builds a **relationship between thread-local invariants and the global shared state** of the program. The top level even intensifies the global state keeping of VerCors and adds a **history** about data updates to the verification. By this, functional correctness properties can be verified. This is a great achievement, since the verification is happening on thread-level, but still, global properties are verifiable. This is both efficient and practical. All three layers are differently encoded into annotated Silver files for Silicon.
Due to the partition into different layers, the verification with VerCors is kept clear and transparent. The use of wellreasoned program logics makes it to a sound and efficient verification tool. Note that VerCors checks for **partial correctness**, but can not prove termination properties. Only if the given program terminates, it satisfies its global postconditions. This might look inconvenient at first glance, since embedded systems are designed as *non-terminating* in general. However, local properties of *parts of the execution* of an infinitely running thread can still be proven by VerCors, for example, pre- and postconditions of functions or loop invariants. They suffice for many interesting properties of SystemC designs.

3.3.2 Prototypal Verification Language

While VerCors supports Java, C, and more languages, it also has its own Prototypal Verification Language (PVL). PVL is mainly used for research purposes and provides new features of VerCors first. It is **inspired by the syntax and structure of Java**, so it has classes, methods, threads, and more. Note that the termin *method* is used in PVL due to its origin in Java, while its equivalent in SystemC is a *function*, because it is the term used by the C++ community. Technically, functions and methods are the same components of the program structure. We only use different terms to distinguish more clearly between SystemC and PVL code. In contrast to Java, PVL has **no visiblity modifiers** like public or private, only the keyword `static` is used in method signatures.

PVL implements a type system and provides three base types: `int`, `boolean`, and `void`. Further types can be added by implementing new classes. The standard operators to build expressions from integer or boolean values and variables are presented in Table 3.3.

Logical Operators	`&&  ||  !  !=  ==`
Comparison Operators	`<  <=  >  >=`
Arithmetic Operators	`+  -  *  /  ++  --`
Conditional Operator	`b ? e1 : e2`

Table 3.3: VerCors Standard Operators.

Note that the conditional operator takes a boolean value `b`, but returns an expression `e1` or an expression `e2` which can be of any type. The only restriction is that `e1` end `e2` must be of the same type. So even if the conditional operator looks like a short if-statement at first glance, it is an operator to build an expression instead of a control flow statement. VerCors also has control flow statemens which are presented in Table 3.4. For assignments, expression `e` has to match the type of variable `x`.

Assignment	`x = e;`	`x`: variable identifier `e`: expression
If-Statement	`if (b) then {s1} else {s2}`	`b`: boolean `s1`, `s2`: sequence of statements
While-Loop	`while (b) {s}`	`b`: boolean `s`: sequence of statements
For-Loop	`for(int i = e1; b; e2) {s}`	`i`: variable identifier `e1`, `e2`: integer expression of `i` `b`: boolean `s`: sequence of statements
Return	`return e;`	`e`: expression of method's return type

Table 3.4: VerCors Control Flow Statements and Assignment.

To create a branching, an `if`-statement can be used. Loops are also integrated in PVL. The `while`-loop repeats a sequence of statements `s` as long as the boolean condition `b` ist fulfilled. If a more specific control variable is requested, a `for`-loop can be used. The branching and loop statements behave as generelly expected; their syntax is shown in Table 3.4.
The last presented expression is the `return`. The statement terminates the control flow within a method implemented in PVL by passing a return expression's value matching the method's return type. To get a more general view of class and method structures in PVL, we provide an example in Listing 3.12.
Like in Java, a class has a name, any number of fields, any number of constructors and optional further methods. Besides the base types, also arrays or objects of other classes can

```
1  class ClassName {
2
3    // Fields
4    boolean isField;
5    int[] integerArray;
6    AnotherClass objectOfClass;
7
8    // Constructor
9    ClassName(int arraySize) {
10     // this denotes the current object.
11     this.isField = true;
12     this.integerArray =  new int[arraySize];
13     this.objectOfClass = new AnotherClass();
14   }
15
16   /* Method:
17     With a multiline comment.
18   */
19   void setNumber(int m) {
20     integerArray[0] = m;
21   }
22 }
23
24 class AnotherClass {
25   // Empty Class
26 }
```

Listing 3.12: VerCors Class Structures Example.

be declared as fields like in lines 5f. In contrast to Java, PVL files can contain more than one class, which do not need to be subclasses of another. Corresponding, the file name is not necessarily required to match a class name.

Arrays can be of any type `T`—base type or self-defined classes—and get initalized by the expression `new T[i]`, where `i` is a non-negative integer value. The values of an array can be accessed via their indices as shown in line 20.

Objects of a Class `C` are initalized by calling its constructor `new C(...)`, providing necessary arguments in parentheses. To refer to the current object, the keyword `this` is reserved. It can also be used to distinguish between a method's argument and a class's field. The keyword `null` specifies that some variable is not set.

Methods in PVL can take any number of arguments and define one return type. If no value should be returned by the method, `void` is declared as the return type in the method's signature. At any point in a PVL file, single-line or even multi-line comments can be added. The explained PVL syntax is an excerpt handling the relevant statements for this work. For a more detailed description, we recommend to have a look at the PVL Syntax Reference on the VerCors's website [Ver21].

If we called VerCors to verify our example class in Listing 3.12, it would complain, because we did not define any permissions about `integerArray` (line 5), although we modify it in the `setNumber` method. So even if we do not write specific properties for verification, we must

provide specifications of the program's semantics for VerCors. Therefore, we introduce the Specificiation Language within the next subsection.

3.3.3 Specification Language

First, we clarify that the **Specification Language is independent of PVL**. The described features are supported for all input languages: PVL, Java, C, OpenCL, and OpenMP. In this work, we add specifications to PVL code only, but they could also be used in a program written in any of the other input languages.
The style of specifications in VerCors is inspired by the Java Modeling Language (JML) [LBR98]. Specifications consist of boolean expressions which are introduced by specification keywords and finished with a semicolon. For "real" input languages like Java or C, specifications need to be put in comments starting with an `@`. In PVL, specifications can be directly written into the code.

Assumptions and Assertions

The most basic types of specifications are **assumptions** and **assertions**. VerCors can prove that an assertion holds under the assumption of a specific state. In particular, the assumption does not even need to match the real program state. We give an example in Listing 3.13.

```
1 int x = 0;
2 assume x == 1;
3 int v = x + 1;
4 assert v == 2;
```

Listing 3.13: VerCors Assumption and Assertion Example (Pass).

Although the real value of `x` is `0` in line 2, VerCors can assume it to be `1` instead. Under this assumption, the assertion `v == 2` holds. If we remove line 2, as in Listing 3.14, the verification by VerCors fails.

```
1 int x = 0;
2
3 int v = x + 1;
4 assert v == 2;
```

Listing 3.14: VerCors Assumption and Assertion Example (Fail).

Even if specifications need to be side-effect free to keep the program state correct, wrong assumptions can influence the results. Assumption and assertion in combination behave like a logical implication. If an *unsatisfiable assumption* is made, then *everything* can be concluded ("ex falso quodlibet"). For example, the program in Listing 3.15 always passes, no matter what assertions are made. This is why **assume** can help to find a missing precondition, but should not be used for the general verification of specifications.

```
1 int x = 0;
2 assume (x == 0 && x != 0);
3 int v = x + 1;
4 assert v == 2;
```

Listing 3.15: VerCors unsatisfiable Assumption Example (Pass).

However, assertions stated with they keyword **assert** are helpful to give VerCors extra knowledge about the intermediate program state. Sometimes, difficult specifications can be proven more easily by providing more intermediate results. In particular, the concept of assumptions and assertions is similar to preconditions and postconditions of methods.

Preconditions and Postconditions

Preconditions of a method define in what state the program is required to be to have a valid call of the method. They are introduced by the keyword **requires**, as shown in Listing 3.16. Accordingly, postconditions specify the program state after the execution of the method. They are denoted by the **ensures** statement. Specifications holding before and after the method's execution can be stated with the **context** keyword. It is convenient to use them for permissions and null checks.

```
1 requires x == 25;
2 ensures v == 42;
3 void prePostExample(int x) {
4   v = x + 17;
5 }
```

Listing 3.16: VerCors Pre- and Postcondition Example.

VerCors assumes the preconditions and checks whether the postconditions can be deduced from the method body. So the postconditions specify a method's behaviour, and VerCors checks whether it behaves as expected. This **verification is performed locally by VerCors**. That means that it does not include a check where the method is called, but only verifies the postcondition unter the assumption of precondition and the method implementation. The verification of the method's preconditions takes place in the verification of the method of the caller.
For example, if a method calls `prePostExample(25)`, VerCors would perform the preconditions' check while verifying the calling method. The verification of `prePostExample()` happens separately. As long as the method is not called somewhere, every precondition is valid, since it has never been asserted somewhere. In Listing 3.17, we give an example which passes, although the precondition is always `false`. However, since we know from the principle of explosion in mathematical logics, any assertion can be deduced from a wrong assumption. Only when another method would call `unsatisfiablePreconditionExample()`, VerCors would complain about the unsatisfiable precondition.

```
requires false;
ensures 17 == 42;
void unsatisfiablePreconditionExample() {
  // Do anything.
}
```

Listing 3.17: VerCors Unsatisfiable Precondition Example (Pass).

The modularity of the verification is one of VerCors's key concepts. By dividing the full program verification into local verifications of methods, the verification performance increases significantly. Pre- and postconditions together build a contract, which must be satisfied by the method's implementation. That is why verification performed with VerCors is called **contract-based**.
Note that the order of pre- and postconditions is important for VerCors. For example, postconditions can not be specified before preconditions. It is also possible to refer to a field's value before the method's execution via the **old**`(expr)` statement. We already gave an example of this in Listing 3.11. If the method does not return `void`, but has a return value, specifications about this value can be made with **\result**.

Loop Invariants

Until now, we presented simple program examples with sequential statement flows. If more complex constructs like loops are used, further specifications are necessary to enable the verification. VerCors performs a static analysis, therefore it does not necessarily know how often a loop is executed. However, this is rather a benefit than a disadvantage, since VerCors can prove the general behaviour specification of loops. Specifications which hold directly before entering and after evaluating one loop execution are called **loop invariants**. They are declared in VerCors by the **loop_invariant** keyword.

```
requires a > 0 && b > 0;
ensures \result == a*b;
int mult(int a, int b) {

  int res = 0;

  loop_invariant res == i*a;
  loop_invariant i <= b;
  for (int i = 0; i < b; i++) {
    res = res + a;
    }

  return res;
}
```

Listing 3.18: VerCors Loop Invariant Example [Ver21].

In Listing 3.18, we give an example of a multiplication performed within a loop. Our loop invariants in lines 7 and 8 specify that the variable `res` computes the (intermediate) product of `a` and `b`, and that the loop terminates correctly. The loop invariants hold before entering the loop, because both `res` and `i` are initialized with zero and `b` is required to be greater than zero by the method's precondition. Furthermore, they still hold after a loop execution, because `i` is incremented and `a` added to `res` once during one loop execution. Under the assumption that `res == i*a` holds before a loop execution, it still holds after a loop execution.
Due to the loop condition, VerCors knows that the loop exactly terminates if `!(i < b)` holds. Together with the loop invariant `i <= b`, VerCors concludes that `i == b` must hold after the termination of the loop. And since the invariant `res == i*a` has been preserved by every loop iteration, the combined knowledge of it and `i == b` enables VerCors to assert `res == a*b` when `res` is returned by the method. By this, the precondition `\result == a*b` can be verified.
With regard to their behavior, loop invariants can be seen as pre- *and* postconditions of one loop evaluation. Together with the base check for entering the loop, the proof structure is similar to a proof by mathematical induction. Like for the induction step, it is important that the loop invariants hold before and after an *arbitrary* execution step. So again, VerCors can prove this locally without knowing every possible value of the involved variables.
Still VerCors *can not* prove the termination of the loop. It asserts all specifications under the assumption that the loop terminates (partial correctness).
Furthermore, the writing of necessary loop invariants can be very challenging, if the loop implementation is more complex. The automatic creation of loop invariants is difficult and a highly active field of research.

Permissions, Resources and Predicates

So far, our examples reasoned about primitive stack variables, so we did not care about the access to these variables[3]. Though real programs do not only manage stack variables, but also variables located on the heap, which is **shared memory**. For the use of heap variables, VerCors requires to define explicitely which thread has read or write permissions. In Section 3.3.1, we described the logical foundation of permissions, now we explain how they are implemented in the Specification Language.
Technically, permissions are encoded by a new data type called **Resource**. A resource behaves like a boolean expression except for the fact that it can contain permissions as well. We reuse our example from Listing 3.18, but replace the stack variables `a` and `b`—formerly passed by arguments to the method—by class fields, which are stored on the heap. The resulting program is shown in Listing 3.19.

[3]Precisely, some examples already involved permissions, but we left them out for simplicity. However, for the sake of completeness, the listings' source files in directory `/listings` contain the complete working code.

Since `a` and `b` are accessible outside of the method `mult()` now, the caller of `mult()` must have permissions for `a` and `b`. They are specified in lines 5 and 6. For both class fields, at least read permissions are required by the statement **Perm**`(heapVar, frac)`. They are denoted by a `frac` greater than zero and less than one. If `frac` is equal to one, the call of `mult()` would require exclusive write permissions, like stated for `b`. In this case, no other thread could access the heap variable during the method's execution. For field `a`, other threads can read it simultaneously with the read by `mult()`.
`frac` is also a new data type introduced for specifications. Its syntax differs from a numerical fraction, because it is written with a backslash; for example `1\2` and *not* `1/2`. Permissions can either be stated by one permission per line or combined within one line by the seperation conjunction `**`, exemplified in line 8.

```
1  class Permissions {
2
3    int a, b;
4
5    requires Perm(a, 1\2);  // Read Permissions
6    requires Perm(b, 1);  // Write Permissions (exclusive)
7    requires a > 0 && b > 0;
8    ensures Perm(a, 1\2) ** Perm(b, 1);
9    ensures \result == a*b;
10   int mult() {
11
12     int res = 0;
13
14     loop_invariant Perm(a, 1\2) ** Perm(b, 1);
15     loop_invariant res == i*a;
16     loop_invariant i <= b;
17     for (int i = 0; i < b; i++) {
18       res = res + a;
19     }
20
21     return res;
22   }
23 }
```

Listing 3.19: VerCors Permissions Example.

Permissions required by a method call *must* be ensured after the termination of the method, so that the permissions are passed back to the caller thread. For example, if **Perm**`(b, 1\2)` would be specified instead of **Perm**`(b, 1)` in line 8, a **Permission Leak** is created. It would never again be possible to establish **Perm**`(b, 1)` at another point in the program, since `1\2` fraction of the permissions are lost.
In addition, **the order of permission statements and general specifications is important**. Heap variables can only be used in specifications, if permissions for them have been stated *previously*. So moving line 7 before line 5 lets the program verification fail as well.
Permissions are not only required for the call of a method, but must also be added to the loop invariant, if the fields are used within the loop. By dealing with permissions, the lines

of specification code increase largely. Though, to make the specifications less duplicate, more modular and transparent, predicates can be used.
A **predicate** can sum up or abstract specification statements. It is written like a function with return type **resource**, but can only be "called" in specifications. We give an example in line 4 of Listing 3.20. In this example, the predicate does not have any arguments, but we could also pass `a` or `b` as an argument to `fieldPerms()` instead of addressing the fields directly. By using this predicate, we have to write detailed permissions only at one location in the code.

```
1  class Predicates {
2
3    // Definition of inline predicate
4    inline resource fieldPerms() = Perm(a, 1\2) ** Perm(b, 1);
5    int a, b;
6
7    requires fieldPerms() ** (a > 0 && b > 0);
8    ensures fieldPerms() ** \result == a*b;
9    int mult() {
10
11     int res = 0;
12
13     loop_invariant fieldPerms() ** res == i*a ** i <= b;
14     for (int i = 0; i < b; i++) {
15       res = res + a;
16     }
17
18     return res;
19   }
20 }
```

Listing 3.20: VerCors Predicates Example.

In this work, we will only use **inline predicates**, which means that the stated specifications of the predicates will be inlined, whereas the predicates are called before verification. If no **inline** keyword is added, we would have to explicitely *fold* (resp. *unfold*) the predicates. This is more complex, but not used for our case studies, so we refer to the VerCors Wiki for more details.
We simplified our multiplication in Listing 3.20 even further by summing up all preconditions in one line 7. Boolean expressions like `(a > 0 && b > 0)` can be conjuncted with resources by the separation conjunction `**`. The full term is now of type **resource** instead of `boolean`. Note that still *boolean* values could be conjuncted by the boolean conjunction `&&`, but only conjuncted with resoures with the separation conjunction. So the parentheses in line 7 are necessary, otherwise VerCors would evaluate `fieldPerms() ** a > 0` first, but fail to conjunct the result with `b > 0` by `&&` afterwards. It is also possible to conjuct multiple boolean values one by one with a resource by the separation conjunction. An example is shown in line 13 for the loop invariant.

In examples up to now, only primitive fields have been declared. If a program contains an array, the reasoning about permissions of this array gets more extensive. **Every element of an array is treated as a seperate location of the heap**. That means every time an array element is accessed, the permission must not only be stated for the array field, but also for its element's field. In addition, this is only possible, if the array has been initialized and the element's index is within the array's range.

```
1  class arrayIndexing {
2
3    int[] array;
4
5    requires Perm(array, 1\2);
6    requires array != null && array.length > 0;
7    requires Perm(array[0], 1);
8    void arrayPermissionExample() {
9      array[0] = 42;
10   }
11 }
```

Listing 3.21: VerCors Arrays and Permissions Example.

In Listing 3.21, we show an example. At least read permissions are necessary to use `array` in the specifications. Furthermore, `array` must not be null and its length greater than zero. Only after this specifications, VerCors can reason about the permissions of the array's first element `a[0]`. Furthermore, the caller of `arrayPermissionExample()` must have write permissions for `array[0]`, otherwise the assignment in line 9 would fail.
Depending on the program's complexity, it is easier to write **Perm**`(array[*], 1)` for requesting write permissions for *all* elements of the array instead of explicitely stating permissions for only the used element. The expression **Perm**`(array[*], 1)` is "Syntactic Sugar" and gets rewritten to a quantified specification internally by VerCors. We describe quantifiers in a part of Section 3.3.3 later on.

Axiomatic Data Types

Specifications must be side-effect free. This is necessary to guarantee that appending of specifications does not change program semantics. For this, **axiomatic data types** are a convenient feature of the Specification Language. In our work, we focus on the data type `seq<T>`. It represents a sequence of elements of any type `T`. Like lists, sequences have an order, and are immutable.
In Listing 3.22, we show how to create and use a sequence. The initialization is very similar to arrays. However, the length of a sequence is differently denoted by `|s|`. It is possible to access the sequence's elements directly via an index, but *not* to assign new values directly. So an assignment like `startSeq[0] = 7` would fail. Still, sequences can be concatenated or are combinded to a new sequence by *appending* new values. From an object-oriented view,

```
1  // Initalization
2  seq<int> startSeq = seq<int> {};
3  seq<int> finalSeq = seq<int> {17, 42};
4
5  // Check for Length
6  assert |startSeq| == 0;
7
8  // Indexing and Equality of Sequence Element
9  assert finalSeq[0] == 17;
10
11 // Concatenation
12 startSeq = startSeq + finalSeq;
13
14 // Check for pairwise-equal Elements and Order
15 assert finalSeq == startSeq;
16
17 // Appending of Value
18 startSeq = startSeq ++ 7;
19 assert startSeq[2] == 7;
```

Listing 3.22: VerCors Sequence Example.

these resulting sequences are completely new (immutable) objects, but *not* the old ones which have been modified.
Sequences facilitate the formulation of specifications. While we have to care about arrays being not `null`, specifying explicit array lengths and handling permissions of the array elements, sequences do not produce this overhead. They can be directly used for equality checks and more. So we will use sequences for easing our case study's specifications instead of encoding everything into arrays.

Quantifiers and Logical Implication

In general, specifications are composed of the same basic expressions as the original input language, in our case PVL. In addition, some further expressions have been added to the Specification Language to facilitate the specification writing. We use the implication operator and quantifiers in our case studies and present their syntaxes in Table 3.5.

Logical Implication	`==>`
Universal Quantifier	`(\forall varDecl; cond; expr)`
Existential Quantifier	`(\exists varDecl; cond; expr)`

Table 3.5: VerCors Quantifiers and Logical Implication.

The implication operator takes two boolean expresses and behaves as the conventional logical implication. It enables reasoning under the assumption of specific assertions and enables transparent case distinctions.

Quantifiers are helpful to write specifications about multiple elements at once, for example, array or sequence elements. Especially, **quantifiers can still reason about all elements of the array without knowing its exact size**. Similarly to a loop declaration, a quantifier has a control variable declared first. It is followed by a condition restricting the range of the variable. The third part is the expression to reason about the elements. A basic example would be

```
(\forall int i; 0 <= i && i < array.length; array[i] != 0)
```

which states that all elements of `array` are not zero. Also the syntactic sugar **Perm**`(array[*], 1)` for permission reasoning is rewritten internally to a quantified expression, precisely:

```
(\forall* int i; 0 <= i && i < array.length; Perm(array[i], write))
```

Besides the universal quantifier, also an existential quantifier is implemented in the Specification Language. So we could also reason whether there exists an element in the array which is not zero. Quantifiers can also be nested, but every nesting increases the difficulty of verification. VerCors has to reason about every possible value of `array[i]`, if it is contained in specification properties. VerCors may verify a program once, but another verification call on the same program may not even terminate. So in general, quantifiers should be cautiously used in VerCors.

3.3.4 Concurrency in VerCors

In the previous sections, we explained the base concepts and syntaxes of PVL and the Specification Language. Now, with a general understanding of verification with VerCors, we focus on concurrency concepts. Handling of concurrency or even parallelism with VerCors is not yet widely explained in its wiki, but the publications by Haack et al. [Haa+14] and Blom, Darabi, and Huisman [BDH15] describe the logical foundations and their encodings into VerCors.
Haack et al. [Haa+14] focuse on the verification of iteration contracts for loop parallelism, while Blom, Darabi, and Huisman [BDH15, esp. chapters 3f] introduce the adaptation of CSL to a multithreaded language and is highly relevant forthe subsequent sections.

Multithreading with PVL

PVL supports different multithreading concepts. For our research, we use the concept of the **Fork-Join Model**. It realizes dynamic thread creation. First, only one sequential master thread exists, but several task threads can be dynamically created by the master thread. New threads can also spawn more threads. This creation of task threads is called **Fork**. Afterwards, the task threads execute in parallel and work with shared memory. So it is necessary to handle read and write permissions for heap variables to avoid data races. If no more execution instructions are left, the task threads synchronize with each other and

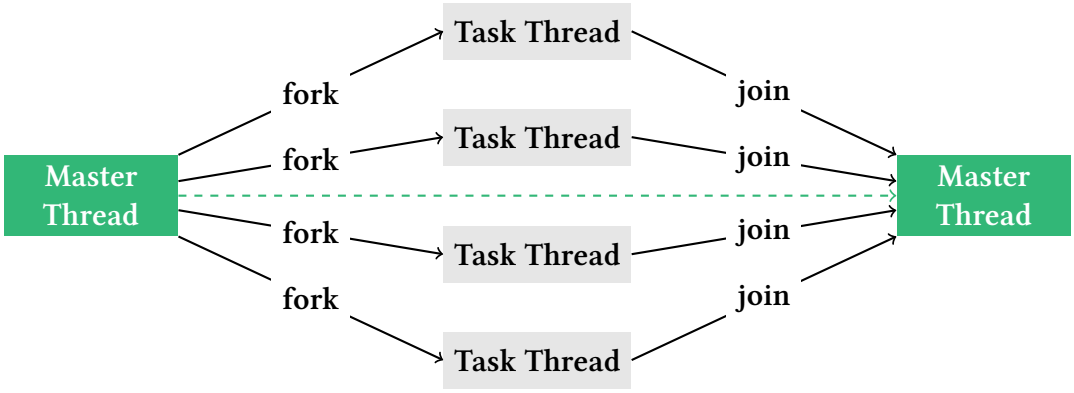


Figure 3.8: Fork-Join Model.

terminate, so they are *joined* with the thread that calls **Join**. We show an illustration of the Fork-Join model in Fig. 3.8.
However, PVL does not have a specific master thread, but instead its statements `fork` and `join` can be simply called within one of the program's methods. The syntax is presented in Table 3.6. A `thread` is not a particular data type, but an object whose class implements the method `void run()`. By this, VerCors will assume the preconditions of the thread's `run()` method after the call of `fork`. Furthermore, permissions are passed from the master thread to its task threads as specifications by the `run()` method's preconditions when the task threads are forked.

Create and Start Thread	`fork thread;`
Synchronize and Terminate Thread	`join thread;`

Table 3.6: VerCors Thread Intialization and Termination Statements.

While VerCors actually does not execute the PVL program for verification, it still assumes any possible execution order of the threads' instructions. So both **real parallelism as well as interleaving concurrency are considered** for reasoning about the program's specifications. If the `join` statement is called, the postconditions of the `run()` are ensured. In particular, all permissions stated in the postconditions are given back to the master thread. The absence of data races and memory safety during execution of the task threads is guaranteed by the correct definition of permissions. Permissions can be transferred between threads during their execution. Moreover, multiple threads can read the same heap memory simultaneously, but only one thread is allowed to write. This is encoded by the fractional permissions based on CSL.

Synchronization of Threads

To pass permissions or to synchronize threads without finally terminating them, **Monitors** are introducted to PVL. A monitor comprises critical execution parts, which should not be occupied by multiple threads at the same time, for example, a write assignment. A thread

which *enters* the monitor owns it, so that no other thread is able to enter the monitor at the same time. After the owner of the monitor *leaves* it, any other thread can become the new owner of the monitor.
To enter a monitor, a thread in PVL has to acquire a **Lock** by the statement `lock lockObject`. For leaving the monitor, it has to release the lock. This is denoted by the statement `unlock lockObject`. All specifications and information which should be exchanged between threads, must be encoded in a **Lock Invariant**. They are asserted upon acquiring a lock and proven upon releasing it. Furthermore, if the critical execution part contains a loop, the ownership of the lock must be added to the loop invariant. This is specified by the statement **held**`(lockObject)`. An overview of synchronization statements and specifications is given by Table 3.7.

Aquire Lock	`lock lockObject;`
Release Lock	`unlock lockObject;`
Lock Invariant	**lock_invariant** `expr;`
Lock Ownership	**held**`(lockObject);`
Wait for Lock Ownership	`wait lockObject;`
Notification for Released Lock	`notify lockObject;`

Table 3.7: VerCors Thread Synchronization Statements (PVL and **Specifications**).

Due to locks, the concept of **mutual exclusion** is integrated into PVL. However, in general, monitors also have the functionality of **cooperative scheduling**. This concept is also implemented by PVL, but it is not the same cooperative scheduling as in SystemC. A thread can suspend its execution by calling a `wait lockObject` statement and release the ownership of a monitor to another thread. Technically, the lock is released and the lock invariant gets reestablished.
If another process calls the `notify` statement, the waiting thread can resume its execution by acquiring the released lock. To keep correctness prescribed by the programming language semantics, a call of `notify` lets VerCors check whether the calling thread indeed owned the lock. Furthermore, if a thread calls `notify lockObj`, *all* waiting threads could potentially proceed and compete to acquire the released lock.

3.3.5 Main Advantages of VerCors Tool Suite

In the previous sections, we described the VerCors tool architecture, PVL, and the Specification Language in detail. Analogously to the SystemC chapter, we sum up why VerCors is a suitable choice for our research purpose.
VerCors's beginnings reach back around ten years, but have been focusing on concurrency concepts from the bottom up. While other theorem provers only handle sequential input languages, or only selected concurrency-enabling languages, VerCors pursues a **language-independent** and general approach of concurrency verification. This makes it applicable

for a wide range of interesting verification problems. On top of that, VerCors does not only perform its verification modularly, but also has a **modular tool architecture**. Instead of creating a completely new verification tool, it uses well-established backend tools. VerCors itself focuses on a sophisticated use of powerful solvers, but keeps the verification interface for input programs extendable and transparent. To sum up these advantages:

1. VerCors performs **static analyses** and its verification is designed as modular, so **it scales very well for large state spaces**, which is a weakness of all model checking approaches.
2. VerCors can not only handle specific concurrency-modeling languages. Moreover, it is designed to be **language-independent**.
3. VerCors is based on a very **powerful backend**: Both Viper [Vip21] and the Z3 SMT Solver [DB08] are well-known in the formal methods community and among the leading tools in their research area.
4. The logical foundation of concurrency and permission handling is based on a sophisticated combination of Implicit Dynamic Frames [SJP09] and Concurrent Separation Logic [BO16]. It is **sound and well-constructed**.

All in all, we see VerCors as a very promising choice for our approach to verify SystemC designs.

4

Transformation from SystemC Designs to PVL Programs

Our major research objective is to develop a **semantics-preserving transformation** from SystemC designs to annotated PVL programs. By this transformation, we gain formal semantics for the informally defined SystemC design and can verify properties deductively with VerCors. Fig. 4.1 shows an overview of the verification process. Currently, all of the steps are performed manually, but allow more automation in future.

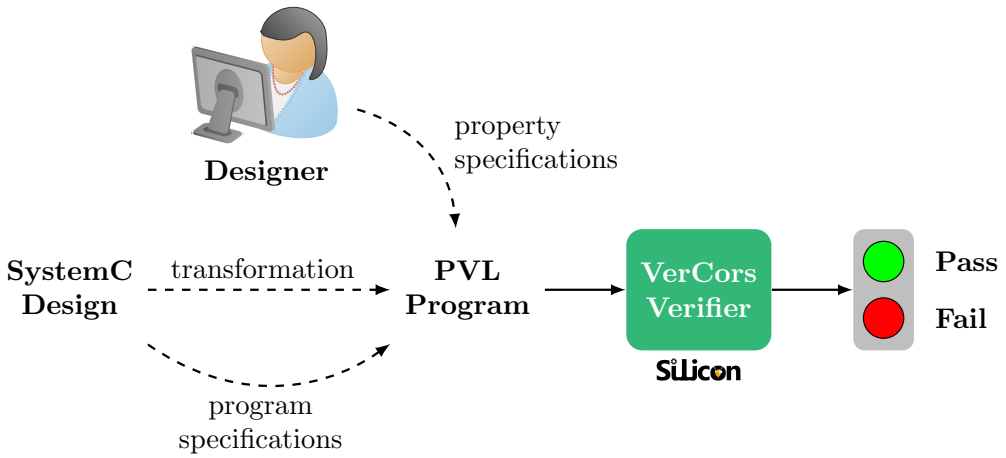


Figure 4.1: Verification Process of SystemC Designs with VerCors.

First, the SystemC code of the design is syntactically transformed into equivalent PVL code. For this, we will present (informal) translation rules for an initial subset of SystemC constructs. Second, permissions for heap variables and further specifications related to the program structure must be added to enable the verification with VerCors. While the addition of specifications produces an extra effort for the transformation, we will outline a

set of specifications, which can be automatically deduced from the program's structure. Lastly, the property specifications must be added manually by the system designer. The resulting annotated PVL program is passed to VerCors, which uses Silicon as its verification backend tool. Silicon verifies the transformed program, either resulting in *Pass* or *Fail*. If the transformation result will pass, we have already **guaranteed data race freedom and memory safety**, and can conclude that **the specified safety properties hold**. If the verification fails, VerCors will provide an error message which assertion does not hold. In this case, the input SystemC design has not been *safe*, either with respect to the handling of permissions, or it could not satisfy the stated safety properties.
A major challenge of the transformation is the implementation of SystemC's cooperative scheduling semantics in PVL. In SystemC, processes can not be interrupted, they only suspend themselves. The event mechanism enables processes to stop their execution at some point, and to continue it later. Thus, the process interleaving is determined by the processes themselves. With VerCors, specifications are verified for *every possible interleaving* of processes. If no extra mechanisms are used, a process could be interrupted *at any time*. Hence, in contrast to SystemC's **non-preemptive** scheduling, PVL assumes **preemptive** scheduling.
To solve this problem, we use the locking mechanism provided by PVL for synchronizations of threads (cf. Section 3.3.4). If a thread wants to modify an object, it needs to own the lock of this object. Hence, no other thread can access the object at the same time (mutual exclusion). This results in our idea to collect permissions for *all heap variables* within **one global lock** object. The global lock is passed between the threads and ensures that no processes are executed simultaneously. Still, an interleaving of the threads is enabled by releasing and reentering the global lock.
To sum up, we contribute to the transformation and verification process by the following achievements:

1. We present **(informal) transformation rules for an initial subset of SystemC constructs**. By following these rules, already simple SystemC designs can be transformed into PVL programs.
2. We introduce a **global lock mechanism, which keeps the cooperative scheduling semantics** during the transformation.
3. We identify a **set of specifications that could potentially be added automatically** to ease verification.

In the following sections, we explain the transformation steps one after one. We start with the transformation rules for SystemC's class structures, i. e., how modules and channels are represented in PVL. Afterwards, we go into the details of the function transformation. In particular, we describe which parts of the sequential code can be directly mapped, or which expressions must be slightly adjusted. Based on this knowledge, we transform normal SystemC functions and functions implementing thread processes into methods in PVL. After the transformation of the class and function syntaxes, we add the necessary global

lock to preserve the non-preemptive scheduling semantics. This is also the section, where most of the new specifications are introduced, so it is the most complex one. Lastly, we give an outlook on how events will be modeled in PVL in the future.
For selected SystemC components, we show the original source code and deduce the result code in PVL. In order to distinguish more clearly between SystemC and PVL code, we use different highlighting colors for SystemC and PVL keywords.

4.1 Supported SystemC Subset

The IEEE SystemC Standard [IEEE11] consists of 638 pages in total. In addition, as a C++ library, SystemC covers the full expressiveness of C++. Our transformation does not support the full SystemC and C++ standard for now; instead we consider a smaller language subset, which still covers relevant real SystemC designs.
While SystemC supports a large amount of **data types**, PVL has three primitive data types: `int`, `boolean`, and `void`. Besides, it offers **arrays**, but has a limitied support for pointers and **dynamic memory management**. Therefore, our transformation can handle `int`, `boolean`, and `void` types, as well as arrays of these types, but no explicit pointer variables. It would be possible to realize some of the more complex SystemC data types by an implementation based of the integer representation in PVL, but we currently do not support this. The VerCors team is already working on a floating point data type, so we would like to extend our approach to this new data type in the future.
The main parts of the actor-oriented design of SystemC can be kept in PVL. We support the general division of processes encapsulated in modules, and communication flow via channels. Though, **hierarchy and scopes** are concepts that are not (yet) implemented by PVL. There exist proposals and ideas how to add inheritence to VerCors [Rub20], but its implementation is future work. So for now, we only support flattened SystemC designs. Furthermore, we assume that **all identifiers are unique** to avoid naming conflicts.
Channels are a particular design structure in SystemC, but can be modeled as shared instances of newly implemented classes in PVL. We remove the abstraction components like interfaces and ports, and simply keep the implementation details of channels. In general, **functions** implementing loops or branching statements are transformed into the corresponding expressions in PVL. Like mentioned above, this is only possible as they contain no other than the supported data types. The switch, goto, break, and continue statements are not included into our transformaton rules yet, but we plan to investigate them in the future.
A SystemC module can hold any number of **thread processes**, but a PVL class can hold only one. Hence, a module with multiple thread processes has to be split up, which increases the tranformation complexity. That is why we describe the base case of a module with only one thread process first, and explain the handling of a module with more than one thread process afterwards. **Method processes** differ from thread processes, because method processes suspend themselves by calling a `wait` statement. Still, they can have a static sensitivity list and get called by the SystemC scheduler, but this holds for thread processes

as well. So, in general, method processes can be simulated by thread processes. Therefore, we restrict our current transformation to thread processes and leave out method processes for simplicity.
We require that the given SystemC design has a static structure only, that means **no dynamic variable or process creation** is involved. Dynamic variable or process creation can be modeled with SystemC, but is only used by a small subset of embedded systems. Therefore, our approach is still applicable for most SystemC designs.
Furthermore, we only support **dynamic sensitivity and no static sensitivity**, since static sensitivity can always be replaced by dynamic sensitivity. Processes are sensitive to events; either explicitly, or implicitly by their sensitivity to channels. The advantage, that a static sensitivity is known before the simulation starts, is not important for the static verification. Every `wait` statement which is related to events from a static sensitivity list, can be replaced by a `wait` statement listing explicitly the sensitivity list's events.
The key feature of SystemC is its discrete-event simulation, which is cooperatively scheduled. In this work, we encode the **cooperative scheduling semantics** into PVL, which is the first significant part of the transformation. The second part is the implementation of SystemC's **event** mechanism in PVL, which is also challenging. PVL does not yet support a concept of events like SystemC. Therefore, the current state of our transformation only supports SystemC designs with one event. If more than one event is involved in the execution, the mapping of events must be specified individually and depends on the concrete SystemC design. However, the VerCors team is already working on a **global lock** mechanism, which uses **conditional variables** for the event realization. We designed our transformation to be extensible to this functionality.
The concept of **time** is a relevant part of the SystemC simulation as well. Since VerCors performs static analyses and proves only partial correctness, it is not dedicated to specify timing behaviors of the program. However, we think that a general handling of events with PVL will enable the managing of time also, since specific points of time can also be seen as events.
All in all, the supported SystemC subset is not exhaustive yet, but we provide an easily extensible, modular and semantics-preserving transformation. Unsupported language components are either rarely used in real SystemC designs, or can be added to the transformation mapping based on future VerCors functionalities. Therefore, we see the current state of our transformation not as a final solution, but rather as an important contribution to a complete semantics-preserving transformation of general SystemC designs.

4.2 SystemC Design Example

We illustrate our transformation steps using a small and clear SystemC design, which is illustrated in Fig. 4.2. It consists of two modules A and B, which communicate via a channel. Module A holds only one thread process, while Module B contains an integer member variable, a function, and two thread processes. The channel implements an interface `channel_if`, and is bound to ports derived from this interface. Furthermore, the channel

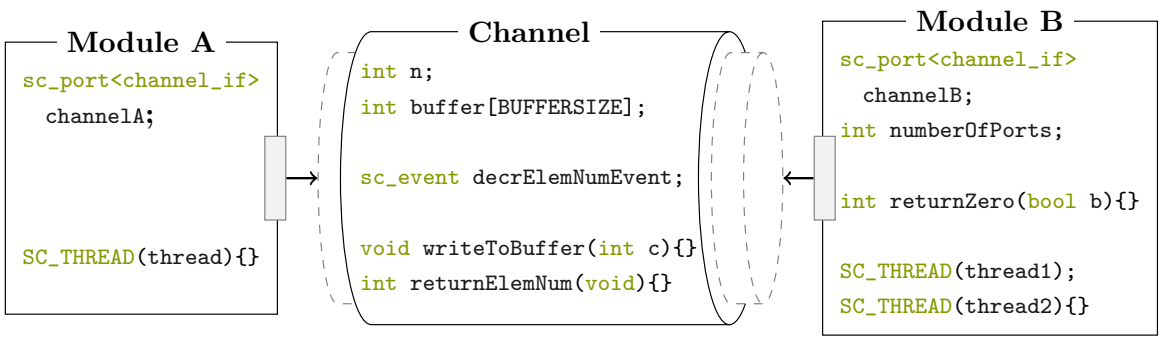


Figure 4.2: General SystemC Design before Transformation.

has an integer variable, an integer array, and two functions. The functions have different combinations of boolean, integer, and void arguments and return types. In addition, the channel contains an event `decrElemNumEvent`. Processes wait for this event when the buffer is full. This is implemented within the `writeToBuffer` function. The event is notified within the `returnElemNum` function.

```
1  int sc_main(int argc, char* argv[]) {
2      // Elaboration Phase
3      channel c_inst("Channel");
4
5      moduleA mA_inst("ModuleA");
6      mA_inst.channelA(c_inst);
7
8      moduleB mB_inst("ModuleB");
9      mB_inst.channelB(c_inst);
10
11     // Simulation of Processes
12     sc_start();
13
14     // No Further Post Processing
15     return 0;   // Simulation Success
16 }
```

Listing 4.1: `main.cpp` of SystemC Design Example.

Listing 4.1 shows the `sc_main` of our example design. In the elaboration phase, two module instances “ModuleA” and “Module B” are declared (Lines 5 and 8), and connected via a channel instance “Channel” (Lines 3, 6 and 9). Afterwards, the simulation is started by calling `sc_start()`. From this on, the SystemC scheduler controls the program execution. For brevity, we do not show the full source code of the SystemC example design in this section, instead it is listed in Appendix A.1.

4.3 Modules and Channels

Modules and Channels are the base of SystemC’s actor-oriented design. We keep the dinstinction of these classes during the transformation to have a transparent result. In

addition, more comprehensive conclusions are possible, if the verification of the resulting program fails.
For the mapping of modules, we have the challenge that a PVL class can only contain one thread, whereas SystemC modules encapsulate any number of thread processes. If we assume the base case with only one thread procss within the module, we directly map the module to a new PVL class exactly named as its SystemC origin, except for a small adjustment. In order to have a clearer distinction, we define as convention that SystemC classes begin with a lowercase letter, and PVL classes with an uppercase letter. For instance, `moduleA` from Fig. 4.2 becomes `ModuleA` in PVL as presented in Fig. 4.3.

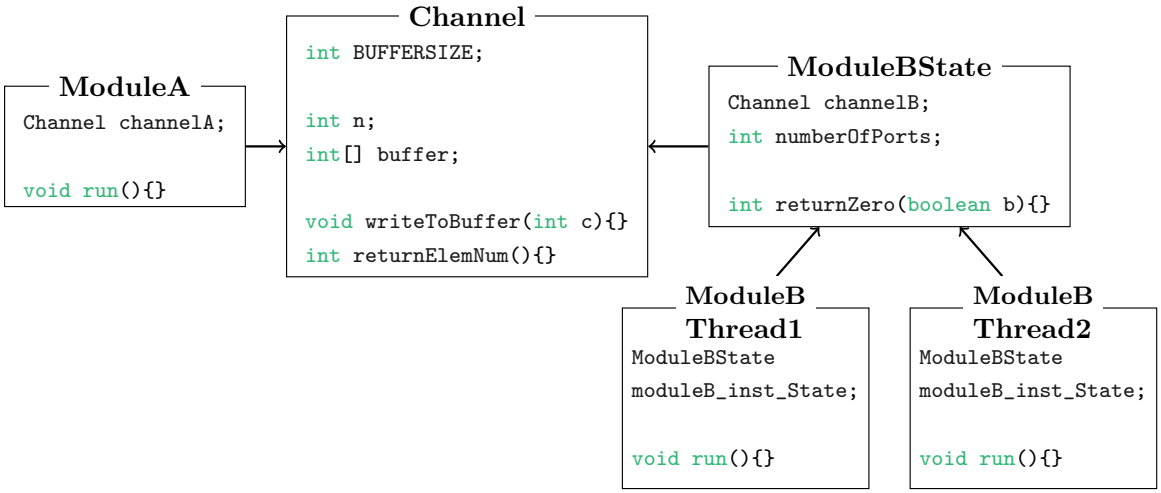


Figure 4.3: PVL Class Structure after Transformation.

Although we declared only one instance per module and channel for simplicity in Listing 4.1, more instances of a module or channel can also be handled by the proposed transformation. The correct mapping of channel and module instances is established by passing the channel instances as arguments to the module instances' constructors. Listing 4.2 shows how a resulting Main method in PVL would look like.

```
1 Main () {
2   Channel channel_inst = new Channel();
3   ModuleA moduleA_inst = new ModuleA(channel_inst);
4   // ...
5 }
```

Listing 4.2: Example Main Method after Transformation to PVL.

Now, we assume the more general case that a module contains more than one thread process. Hence, at least two thread processes have access to the module's member variables and functions. To keep the shared access, we introduce a new class in PVL holding the module's shared state. It is named like the module suffixed with `State`, for example, `moduleB` from the SystemC designs results in a PVL class `ModuleBState`. This shared *state class* contains all member variables and functions of the original SystemC module. In the case of module

B, these are the integer variable `numberOfPorts`, the function `returnZero`, and the channel variable `channelB`.
Each of the module's thread processes is encapsulated into a new PVL class, which is named as the module suffixed with the thread name. Hence, module B's `thread1` and `thread2` result in two new classes named `ModuleBThread1` and `ModuleBThread2`. Both have a member variable holding the same instance of `ModuleBState`. Every new *thread class* implements a method `run()` holding the thread's SystemC statements translated to PVL. More details about the thread process implementation are given in Section 4.5.
A channel of a SystemC design is mapped to a new PVL class named like the channel class in SystemC. It keeps all of the channel's member variables except of events. We describe the event transformation in Section 4.7. Furthermore, all of the channels functions are transformed into PVL. The transformation details of data types and functions are described in the next section.
SystemC components, which only exist for abstraction purposes, are completely removed by the transformation. In particular, this comprises ports or channel interfaces. Since VerCors does not execute the resulting PVL program, there is no advantage in having a separation between computation and communication. Consequently, SystemC variables of type `sc_port<channel_if>` are replaced by an equally-named variable of the channel type, in this case simply `Channel`. Directives declaring values, like `BUFFERSIZE` in the SystemC design example, are transformed into fields, which are initialized in the new class's constructor. Note that SystemC module and channel classes finish with a semicolon, but PVL classes not.
Fig. 4.3 illustrates the resulting class structure of the SystemC design from Fig. 4.2. However, it shows the general component structure, but not how exactly functions are mapped in PVL. This is the next section's topic.

4.4 Functions

In SystemC, functions hold the executable statements of the model. In our transformation, every SystemC function is mapped to a new PVL method. The location of the method depends on the translation rules introduced in the previous section. It is either a directly mapped class, a *thread class* or a *state class*. In this section, we focus on general functions, which do not implement thread processes.

Function Signatures and Data Types

The signature of a SystemC function is taken over; only functions without arguments are slightly adjusted. The keyword `void` must be removed from the argument list in PVL during the transformation. Also the boolean type `bool` in SystemC is mapped to the slightly different named type `boolean` in PVL. Listing 4.3 and Listing 4.4 show the SystemC function signatures of our general model, and their resulting PVL method signatures.
The SystemC type `int` is mapped to PVL's data type `int`. At this point, we have to mention that PVL implements a mathematical interpretation of integers. It assumes that the set of

```
int returnZero(bool b) {
  // ...
}

void writeToBuffer(int c) {
  // ...
}

int returnElemNum(void) {
  // ...
}
```

Listing 4.3: Function Signatures in SystemC.

```
int returnZero(boolean b) {
  // ...
}

void writeToBuffer(int c) {
  // ...
}

int returnElemNum() {
  // ...
}
```

Listing 4.4: Method Signatures in PVL.

integers is infinite, while SystemC's built-in integer type `sc_int` has a fixed numerical range defined by its bit width. We leave the correct handling of arithmetic overflow as subject to future work. The described data type mappings do not hold for function signatures only, but also for every other use of data types, like member variables in the source code.
A special type of a function is the constructor in SystemC, denoted by the macro `SC_CTOR`. It has no return type and takes the class name as its first argument. It is mapped to a constructor method in PVL, whose name consists of the class name only. Listing 4.5 and Listing 4.6 show an example. The SystemC macro `SC_THREAD` is also removed by the transformation, since no processes must be registered at the scheduler any more.

```
SC_CTOR(moduleA) {
  SC_THREAD(thread);
  // ...
}
```

Listing 4.5: Constructor of Module A in SystemC.

```
ModuleA (Channel channel) {
  // No Thread Declarations
  // ...
}
```

Listing 4.6: Constructor of Module A in PVL (Intermediate State).

Besides primitive data types, arrays are also supported by our transformation. Though, their declarations have to be slightly adjusted in PVL. An array of length `i` is declared by `T array[i]` in SystemC, but must be changed to `T[] array` in PVL. The type `T` can be either integer, or boolean. Furthermore, the array in SystemC does not need a separate initialization after its declaration, before values can be written to it. This is different in PVL, where its initialization must be declared explicitely. So for every `array` member variable in SystemC, we add its initialization to the class's constructor by an additional statement `array = new T[i]`.
We illustrate the transformation process in Listing 4.7 and Listing 4.8, which show the comparison between the SystemC channel and the PVL Channel. Up to now, we moved the former SystemC directive to a PVL field, removed scoping and sensitivity keywords, and adjusted the array declaration and the constructor signature.

```
#include <systemc.h>
#define BUFFERSIZE 42

// Interface Definition: channel_if

class channel : public sc_channel,
public channel_if {

  // Internal Data
  private:
  int n;
  int buffer[BUFFERSIZE];

  // Public Events and Functions
  public:
  sc_event decrElemNumEvent;

  SC_CTOR(channel){
    n = 0;
  }




  // Further Functions
};
```

Listing 4.7: Channel Class in SystemC (Excerpt).

```
class Channel {

  // Define Directives
  int BUFFERSIZE;

  // Internal Data
  int n;
  int[] buffer;

  // No Events/Visibilities

  // Permissions and Value Checks
  ensures Perm(n, write);
  ensures Perm(buffer, read);
  ensures buffer != null;
  ensures buffer.length == 42;
  ensures Perm(buffer[*],write);
  Channel() {
    BUFFERSIZE = 42;

    n = 0;
    buffer = new int[BUFFERSIZE];
  }

  // Further Methods
}
```

Listing 4.8: Channel Class in PVL (Intermediate State).

Permissions and Assertions

In order to make our resulting program verifiable by VerCors, it is important to add specifications, like to the PVL constructor in Listing 4.8. We ensure write permissions for every primitive field (Line 13), and read permissions and a null value check for every object field (Lines 14f). If the object field holds an array, as shown in our example, we add a specification of the buffer's length and write permissions for all array elements (Lines 16f). However, the write permission for *all* elements are not necessary, but we provide all of them to keep the transformation more general. Otherwise, we would have to analyze every possible access to an array element in the whole program. This would lead to a strong increase in the transformation complexity. Permissions can only be ensured by the constructor initializing the fields. Afterwards, they are owned by the caller method of the constructor and crucially required, because otherwise no one has ever access to the class's fields. We extend the specification statements during the upcoming transformation steps.

Function Bodies

SystemC's control statements and assignments are transformed to their natural equivalents in PVL. They slightly differ in the precise syntaxes, which is recorded in Table 4.1. While C++ has a precise distinction of pointers and references, we do not keep this by our

transformations to PVL. The support of pointers by PVL is limited, so we can only handle SystemC designs, where pointers can be replaced by references without changing the program semantics. Under this assumption, function calls with an arrow -> in SystemC are simplified to the syntactic delimiter of a dot . in PVL.

SystemC	PVL	Remarks
`x = e;`	`x = e;`	*No difference*
`if (b) s1`	`if (b) then {s1}`	
`if (b) {s}`	`if (b) then {s}`	`s` is sequence of statements
`if (b) {s1} else {s2}`	`if (b) then {s1} else {s2}`	
`while (b) {s}`	`while (b) {s}`	`s` is sequence of statements
`for(int i = e1; b; e2) {s}`	`for(int i = e1; b; e2) {s}`	*No difference*
`return e;`	`return e;`	
`object->callFunction();`	`object.callFunction();`	Care should be taken, if pointers are dereferenced in function calls.

Table 4.1: Transformation Rewriting Rules for Control Flow Statements.

Arithmetic and logical operators are taken over from the SystemC code. A complete overview of supported operators is listed in Table A.2 of the appendix of this work. Compound operators like `x += 1` must be rewritten to `x = x + 1`, since PVL does not support them, but this adds no technical complexity to the transformation.

4.5 Processes

As described in Section 4.3, a function denoted as a thread process by the macro `SC_THREAD` is transformed to the `run()` method in PVL. Both, the thread's SystemC function and its PVL method must have void as the return type and no arguments. This is expressed by

```
void thread(void) {...}
```

in SystemC, and

```
void run() {...}
```

in PVL. In many SystemC designs, a thread process in SystemC is embedded into a `while(true)` loop and shall not terminate. This is preserved by the transformation to PVL and causes no problems, since VerCors does not check for termination, and can still verify specifications locally.

If a thread process is the only one of its module, for instance module A's `thread`, there are only a few adjustments of its resulting PVL class necessary. While channels are bound to ports within the `sc_main` in SystemC, they are simply passed as an argument to the module class's constructor in PVL. All other member variables and functions are preserved.

If a SystemC module contains more than one thread, like module B's `thread1` and `thread2`, the transformation to PVL requires more effort. Instead of a direct access to its channel,

the thread class `ModuleBThread1` has a field for the shared instance of class `ModuleBState`. The state instance holds module B's member variables, like the channel, and member functions. Therefore, the state object—and not the channel object—is passed to the thread class via its constructor's argument. Listing 4.9 shows the short and rather simple excerpt of the original SystemC module B with its thread process. In contrast to this, the resulting PVL class presented in Listing 4.10 is more complex. All usages of module B's variables, and calls of its functions, must be rewritten to the state class instance instead. This is shown in lines 17 and 18 of Listing 4.9.

Furthermore, the necessary specifications added to the PVL code make it much longer. First, the constructor requires the given state object instance to be not null (Line 7). In addition, read permissions must be ensured for the resulting field (Line 9). It is also specified to be not null (Line 10), and must hold the same object instance as passed by the argument after execution of the constructor (Line 11). The combination of null checks, permissions, and equalities of values is a standard construct, which we always have to add as specifications for all non-primitive constructor arguments and class fields.

The permissions for the state object and its fields must be kept during the whole execution of the thread. Otherwise, the access to module B's buffer element in line 26 would fail, because the thread would have *insufficient permissions* to do this. The reason is, that VerCors proves assertions locally, and the permissions have to be owned by the thread before entering the loop and after each iteration. They are ensured by the thread's constructor, but this is not known by VerCors, since it verifies the constructor and the `run()` method separately.

To still pass the permissions between different methods executed by the thread, they could be either specified by the method's preconditions or as a lock invariant. To preserve the non-preemptive scheduling semantics, we have to add a lock to every `run()` method, as this is the easiest possiblity to pass permissions. For methods, which are not implementing a thread, but are called by it, we specify the permissions as pre- and postconditions. The explicit definition of the ownership of permissions is a benefit of VerCors, since data races would never pass the verification.

So far, we presented transformation rules for classes, processes, and methods down to single statements. We illustrated the rules by a transformation of the components of our example SystemC design, but did not preserve the non-preemptive scheduling semantics yet. We solve this remaining problem by introducing a global lock, which establishes the correct execution order of the threads.

```
SC_MODULE(moduleB) {

  sc_port<channel_if> channelB;

  // Variables, Member Functions...

  // Constructor
  SC_CTOR(moduleB) {
    // Process Registration
    SC_THREAD(thread1);
    // ...
  }

  // Interruptable Process 1
  void thread1(void) {
    while(true) {
      returnZero(true);
      channelB->writeToBuffer(42);
      // ...
    }
  }
};
```

Listing 4.9: Module B's Thread1 Process in SystemC.

```
class ModuleBThread1 {

  // Original Fields/Functions moved to ModuleBState
  ModuleBState moduleB_inst_State;

  // NotNull Check of Arguments
  requires moduleBState != null;
  // Shared ModuleBState: Permissions and Value Checks
  ensures Perm(moduleB_inst_State, read);
  ensures moduleB_inst_State != null;
  ensures moduleB_inst_State == moduleBState;
  ModuleBThread1(ModuleBState moduleBState) {
    moduleB_inst_State = moduleBState;
  }

  void run () {
    // Must hold all specifications, conjuncted by **
    loop_invariant true
      ** Perm(moduleB_inst_State, read)
      ** Perm(moduleB_inst_State.numberOfPorts, write)
      ** Perm(moduleB_inst_State.channelB, read)
      // ** more specifications added later...
      ;
    while(true) {
      moduleB_inst_State.returnZero(true);
      moduleB_inst_State.channelB.writeToBuffer(42);
    }
  }
}
```

Listing 4.10: Module B's Thread1 in PVL (Intermediate State).

4.6 Non-Preemptive Scheduler

In SystemC, the scheduler has the control over the simulation. Even if it can not interrupt processes, it manages all wait statements and notifications of events. It handles the current process state and possibly moves a process to the *ready* pool again. Such a concept does not exist in VerCors. For the verification of a multithreaded program, it assumes any possible interleaving semantics of the threads's executions. Our idea is to restrict the interleaving of threads in PVL by collecting all permissions of shared heap variables within one global lock.

The Global Lock and Main Class Structure

Fig. 4.4 visualizes the approach for our example SystemC design. To realize the global lock, we implement two new classes with identifiers `Main` and `GlobalLock`. The `Main` class has not a technical functionality, but keeps the transformation more transparent. Its `main()` method initializes the global lock and calls the lock's `simulation()` function.

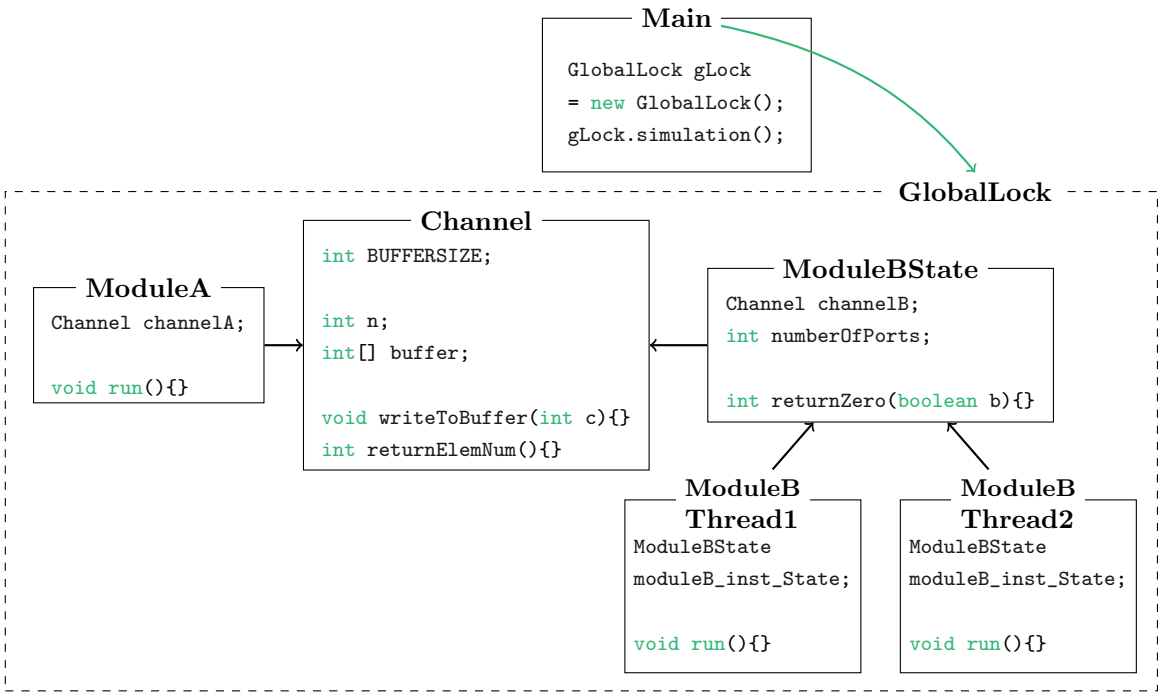


Figure 4.4: PVL Classes after Transformation with Global Lock.

The `GlobalLock` is the maintainer of all thread, state and channel class instances. Therefore, they are declared as its fields and get initialized in the global lock's constructor. The general structure of the global lock is presented in Listing 4.11.
We pass the global lock instance as an argument (`this`) to the constructor of all PVL classes. Otherwise, it would not be possible to formalize specifications of the program state in other classes than the global lock. For our example SystemC design, the classes receiving the global lock via their constructor are the channel, module A, and module B's state and

thread classes (Lines 16 to 20). The method `simulation()` is the transformation equivalent to SystemC's `sc_start()` function. It forks all threads and joins them afterwards.

```
1  class GlobalLock {
2
3    // Fields for Channels, and Module's Threads and States
4    Channel channel_inst;
5    ModuleA moduleA_inst;
6    ModuleBState moduleB_inst_State;
7    ModuleBThread1 moduleB_inst_Thread1;
8    ModuleBThread2 moduleB_inst_Thread2;
9
10   // Permissions of all Objects and Fields
11   resource lock_invariant() = true
12       // ** ...
13   ;
14
15   GlobalLock () {
16     channel_inst = new Channel(this);
17     moduleA_inst = new ModuleA(this, channel_inst);
18     moduleB_inst_State = new ModuleBState(this, channel_inst);
19     moduleB_inst_Thread1 = new ModuleBThread1(this, moduleB_inst_State);
20     moduleB_inst_Thread2 = new ModuleBThread2(this, moduleB_inst_State);
21   }
22
23   void simulation() {
24     fork moduleA_inst;
25     fork moduleB_inst_Thread1;
26     fork moduleB_inst_Thread2;
27
28     join moduleA_inst;
29     join moduleB_inst_Thread1;
30     join moduleB_inst_Thread2;
31   }
32 }
```

Listing 4.11: PVL Global Lock Base Structure.

The main challenge of the global lock is that it must hold *all permissions for all heap variables*, but these heap variables are distributed and shared by multiple threads encapsulated in different classes. For instance, VerCors needs to know whether the channel fields of module A and B are accessing the same heap object, or not. These equalities are semantic knowledge formalized by specifications of the global program state. Their verification with VerCors is challenging, because VerCors's specification language does not implement global invariants. The verification is always done thread-locally, so we must encode global specifications differently.

The particular point of *execution*, where the context is passed between threads, is when the ownership of the global lock changes. Every time a thread acquires or releases the global lock in PVL, the lock invariant is assumed. Therefore, the lock invariant is the place, where we can pass specifications about the global program state between the threads. Hence, we

add a lock invariant to the global lock class, whose basic declaration is presented in Fig. 4.4, line 11. Up to now, it is empty, and VerCors does not know anything about the relation between fields declared in separate classes. We describe the necessary specifications for a successful verification in the next section.

Specifications for Global Program State in Lock Invariant

The complete lock invariant is composed of read and write permissions, null checks, and equality checks. They can be derived from the original SystemC design and the already transformed PVL result classes by performing the following steps:

1. **Read Permissions for Fields of Global Lock:** For every field of the global lock class, i. e., channel, state and thread classes of the transformed PVL program, read permissions must be specified. Otherwise, they could not be used to formalize the global program state.
2. **Read Permissions and Equality of Global Lock:** The global lock instance, which is passed as argument to the classes's constructors, must be the same for all classes of the program. Hence, for every class, read permissions for its field `gLock` must be added. Also the equality of all global lock instances of the classes must be specified. By this, VerCors knows that all classes are related by the same global lock.
3. **Read Permissions and Equality of State Classes:** For all PVL thread classes derived from a SystemC module with multiple threads, read permissions of their fields holding the shared state instance must be added. Then, the equality of all shared state instances must be stated, so that the thread instances are truly working on the same shared heap memory.
4. **Read Permissions of Channels:** For every channel, read permissions of the corresponding channel field of thread/state classes must be added to the lock invariant. To ensure, that the threads are really operating via the same channel instance, the equality of the field's values must be specified.
5. **Permissions, Null and Length Checks of further Heap Variables:** For all not yet handled fields of the classes, write permissions for primitive fields, and read permissions plus not null checks for fields holding arrays and objects are specified. For array fields, write permissions for its fields and a specification of its length are necessary as well.

Since every heap variable produces at least one specification, the lock invariant grows fast. As shown in Listing 4.12, the lock invariant of our small example SystemC design is already 41 lines long. The presented steps to build a lock invariant are complete with respect to all variables and objects of a given model. It is possible that the program's lock invariant does not need to contain all of these specifications to be verifiable. A more detailed analysis of the program's source code could reduce the length of the lock invariant. Hence, the steps 1

to 5 are more general instructions for building a lock invariant to keep the transformation most simple, but possibly less efficient. We see the optimization as future work.

```
resource lock_invariant = true
  // 1. Read Permissions for all Channel, State and Thread Classes
  ** Perm(channel_inst, read)
  ** Perm(moduleA_inst, read)
  ** Perm(moduleB_inst_State, read)
  ** Perm(moduleB_inst_Thread1, read)
  ** Perm(moduleB_inst_Thread2, read)

  // 2. Read Permissions & Equality of the Shared Global Lock
  ** Perm(channel_inst.gLock, read)
  ** Perm(moduleA_inst.gLock, read)
  ** Perm(moduleB_inst_State.gLock, read)
  ** Perm(moduleB_inst_Thread1.gLock, read)
  ** Perm(moduleB_inst_Thread2.gLock, read)
  ** this == channel_inst.gLock
  ** this == moduleA_inst.gLock
  ** this == moduleB_inst_State.gLock
  ** this == moduleB_inst_Thread1.gLock
  ** this == moduleB_inst_Thread2.gLock

  // 3. Read Permissions & Equality of Shared Module States
  ** Perm(moduleB_inst_Thread1.moduleB_inst_State, read)
  ** Perm(moduleB_inst_Thread2.moduleB_inst_State, read)
  ** moduleB_inst_Thread1.moduleB_inst_State == moduleB_inst_State
  ** moduleB_inst_Thread2.moduleB_inst_State == moduleB_inst_State

  // 4. Read Permissions & Equality of Shared Channels
  ** Perm(moduleA_inst.channelA, read)
  ** Perm(moduleB_inst_State.channelB, read)
  ** channel_inst == moduleA_inst.channelA
  ** channel_inst == moduleB_inst_State.channelB

  // 5. Permissions, Null and Length Checks of further Heap Variables
  ** Perm(moduleB_inst_State.numberOfPorts, write)
  ** Perm(channel_inst.n, write)
  ** Perm(channel_inst.buffer, read)
  ** channel_inst.buffer != null
  ** channel_inst.buffer.length == 42
  ** Perm(channel_inst.buffer[*],write)
;
```

Listing 4.12: PVL Lock Invariant for General SystemC Design.

The permissions and specifications required in the lock invariant are a formalization of the global program state. Since this state is not created in the lock invariant, at least the permissions must be ensured by another part of the program. The read permissions specified in Step 1 are ensured by the constructor of the global lock, because it initializes the instances of the channel, state and thread classes. It must also ensure that they are initialized after execution of the constructor. Furthermore, the read permissions and

equality specifications of the global lock are ensured by its constructor. Listing 4.13 shows the resulting specifications of our example SystemC design. Note that there are 32 lines of specifications in comparison to 5 lines of implementation code.

```
1  // Read Permissions and Null Check for Channels, Thread and State Classes
2  ensures Perm(channel_inst, read);
3  ensures Perm(moduleA_inst, read);
4  ensures Perm(moduleB_inst_State, read);
5  ensures Perm(moduleB_inst_Thread1, read);
6  ensures Perm(moduleB_inst_Thread2, read);
7  ensures channel_inst != null;
8  ensures moduleA_inst != null;
9  ensures moduleB_inst_State != null;
10 ensures moduleB_inst_Thread1 != null
11 ensures moduleB_inst_Thread2 != null;
12 // Read Permissions and Null Check for Global Lock
13 ensures Perm(channel_inst.gLock, read);
14 ensures channel_inst.gLock != null;
15 ensures Perm(moduleA_inst.gLock, read);
16 ensures moduleA_inst.gLock != null;
17 ensures Perm(moduleB_inst_State.gLock, read);
18 ensures moduleB_inst_State.gLock != null;
19 ensures Perm(moduleB_inst_Thread1.gLock, read);
20 ensures moduleB_inst_Thread1.gLock != null;
21 ensures Perm(moduleB_inst_Thread2.gLock, read);
22 ensures moduleB_inst_Thread2.gLock != null;
23 // Equality of the Shared Global Lock
24 ensures channel_inst.gLock == this;
25 ensures moduleA_inst.gLock == this;
26 ensures moduleB_inst_State.gLock == this;
27 ensures moduleB_inst_Thread1.gLock == this;
28 ensures moduleB_inst_Thread2.gLock == this;
29 // Status of Thread1 of ModuleB, Thread2 of ModuleB, and Thread of ModuleA
30 ensures idle(moduleB_inst_Thread1);
31 ensures idle(moduleB_inst_Thread2);
32 ensures idle(moduleA_inst);
33 GlobalLock () {
34   channel_inst = new Channel(this);
35   moduleB_inst_State = new ModuleBState(this, channel_inst);
36   moduleB_inst_Thread1 = new ModuleBThread1(this, moduleB_inst_State);
37   moduleB_inst_Thread2 = new ModuleBThread2(this, moduleB_inst_State);
38   moduleA_inst = new ModuleA(this, channel_inst);
39 }
```

Listing 4.13: PVL Constructor Specifications for General SystemC Design.

The global lock's method `simulation()` which forks and joins the threads, must keep all specifications of the constructor. So they are required and ensured as pre- and postconditions, which can be abbreviated by the keyword **context**. The full PVL code including the specifications of the global lock can be found in the appendix, Listing A.6.
All specifications of channel, thread, and state class's fields must be ensured by the constructors of these classes. Listing 4.14 shows the completed constructor of module A after

```
1  // NotNull Check of Arguments
2  requires globalLock != null;
3  requires channel != null;
4  // Global Lock: Permissions and Value Checks
5  ensures Perm(gLock, read);
6  ensures gLock != null;
7  ensures gLock == globalLock;
8  // Field Channel: Permissions and Value Checks
9  ensures Perm(channelA, read);
10 ensures channelA != null;
11 ensures channelA == channel;
12 ModuleA (GlobalLock globalLock, Channel channel) {
13   gLock = globalLock;
14   channelA = channel;
15 }
```

Listing 4.14: PVL Constructor of Module A of General SystemC Design.

the transformation to PVL. It requires the passed global lock and channel instance to be not null. Under this assumption, it ensures read permissions of the corresponding fields, that they are not null, and that they hold the instances given as arguments to the constructor.

Applying the Global Lock on Threads

So far, we described the general idea of the global lock. We explained its implementation and the necessary specifications. Now we apply the global lock mechanism on the threads to achieve the non-preemptive scheduling semantics.

Every thread must be forced to acquire the global lock, before it executes a statement. Since only one thread can own the global lock, multiple threads can never execute simultaneously. If the thread does not release the global lock itself, it can not be taken by any other thread. Hence, the thread scheduling becomes non-preemptive. To implement the acquisition of the lock in a thread, we add the lock acquisition `lock gLock` as the first statement to its `run()` method. The corresponding lock release by `unlock gLock` is placed as the last statement in `run()`. We assume that `gLock` is the identifier of every global lock field.

To acquire the global lock `gLock`, the `run()` method must have read permissions for it, and the global lock must not be null. Therefore, this is required in the method's preconditions. The specifications must also be ensured as postconditions. Hence, they are denoted with the keyword **context**. The equality of the current thread and the instance of the global lock is also specified. Listing 4.15 shows the resulting `run()` method of module A in PVL.

The current state of the `run()` method would not yet pass the verification. VerCors asserts the global lock's invariant when the global lock is acquired, which is no problem. However, it asserts the lock invariant again when the global lock is relased, which is not that easy. The additional difficulty derives from the involved `while(true)` loop. VerCors needs further knowledge about the program state before and after the loop executions, otherwise it can not assert the lock invariant afterwards. That is why we have to encode all specifications

```
1  context Perm(gLock, read) ** gLock != null;
2  context Perm(gLock.moduleA_inst, read);
3  context gLock.moduleA_inst == this;
4  void run () {
5    lock gLock;
6
7    loop_invariant true
8    // ** ...
9    ;
10   while(true) {
11     // ...
12   }
13
14   unlock gLock;
15 }
```

Listing 4.15: PVL Thread Implementation of Module A with Global Lock.

from the lock invariant into loop invariants within a lock area as well.
The global lock instance is the only relation between the other heap variables and the current thread, so it is needed to formalize the loop invariant correctly. Listing 4.16 shows the complete `run()` method of module A of the example SystemC design. Since it is very long, we highlighted the parts of the specifications which differ from the loop invariant in Listing 4.12. The lines 8 to 11 express the permissions, null check, and ownership of the global lock. By the equality in line 21, a relation between the global lock's instance of module A and the current thread can be drawn. All other modifications are only insertions of the prefix `gLock`, because we have a different context in this loop invariant in contrast to the lock invariant. Moreover, the lines 36 and 43 have already been included in the lock invariant without the `gLock` prefix.

```
1  context Perm(gLock, read) ** gLock != null;
2  context Perm(gLock.moduleA_inst, read);
3  context gLock.moduleA_inst == this;
4  void run () {
5    lock gLock;
6
7    loop_invariant true
8      // Permissions and Ownership of Global Lock
9      ** Perm(gLock, read)
10     ** gLock != null
11     ** held(gLock)
12
13     // Read Permissions for all Channel, State and Thread Classes
14     ** Perm(gLock.channel_inst, read)
15     ** Perm(gLock.moduleA_inst, read)
16     ** Perm(gLock.moduleB_inst_State, read)
17     ** Perm(gLock.moduleB_inst_Thread1, read)
18     ** Perm(gLock.moduleB_inst_Thread2, read)
19
```

```
20      // Equality of Global Lock's ModuleA and this
21      ** gLock.moduleA_inst == this
22
23      // Read Permissions & Equality of the Shared Global Lock
24      ** Perm(gLock.channel_inst.gLock, read)
25      ** Perm(gLock.moduleA_inst.gLock, read)
26      ** Perm(gLock.moduleB_inst_State.gLock, read)
27      ** Perm(gLock.moduleB_inst_Thread1.gLock, read)
28      ** Perm(gLock.moduleB_inst_Thread2.gLock, read)
29      ** gLock == gLock.channel_inst.gLock
30      ** gLock == gLock.moduleA_inst.gLock
31      ** gLock == gLock.moduleB_inst_State.gLock
32      ** gLock == gLock.moduleB_inst_Thread1.gLock
33      ** gLock == gLock.moduleB_inst_Thread2.gLock
34
35      // Read Permissions & Equality of Shared Module States
36      ** Perm(gLock.moduleB_inst_State, read)
37      ** Perm(gLock.moduleB_inst_Thread1.moduleB_inst_State, read)
38      ** Perm(gLock.moduleB_inst_Thread2.moduleB_inst_State, read)
39  ** gLock.moduleB_inst_Thread1.moduleB_inst_State == gLock.moduleB_inst_State
40  ** gLock.moduleB_inst_Thread2.moduleB_inst_State == gLock.moduleB_inst_State
41
42      // Read Permissions & Equality of Shared Channels
43      ** Perm(gLock.channel_inst, read)
44      ** Perm(gLock.moduleA_inst.channelA, read)
45      ** Perm(gLock.moduleB_inst_State.channelB, read)
46      ** gLock.channel_inst == gLock.moduleA_inst.channelA
47      ** gLock.channel_inst == gLock.moduleB_inst_State.channelB
48
49      // Permissions, Null and Length Checks of further Heap Variables
50      ** Perm(gLock.moduleB_inst_State.numberOfPorts, write)
51      ** Perm(gLock.channel_inst.n, write)
52      ** Perm(gLock.channel_inst.buffer, read)
53      ** gLock.channel_inst.buffer != null
54      ** gLock.channel_inst.buffer.length == 42
55      ** Perm(gLock.channel_inst.buffer[*],write)
56    ;
57    while(true) {
58      // ...
59    }
60
61  unlock gLock;
62  }
```

Listing 4.16: PVL Thread Implementation of Module A with Loop Invariant.

The loop invariants for `thread1` and `thread2` are equally derived from the lock invariant, so we do not explicitly outline them in this section. Another aspect of the transformation is that all methods, which are called by a thread, must have the invariants' specifications as

its pre- and postconditions. Otherwise VerCors can not verify whether they still hold after the execution of the method. For example, the methods `void writeToBuffer(int c){}` and `int returnElemNum(){}` of the channel class are enriched with the invariant's specifications. The fully transformed PVL program with all invariants can be found in Appendix A.3.
We achieved to implement non-preemptive scheduling semantics by a global lock in PVL. Still, the presented transformation rules can only be applied on a subset of industrial SystemC designs. If more than one event is involved in the process scheduling, our transformation rules can not guarantee that the transformation is fully semantics-preserving. To support the full event mechanism of SystemC, we need an extension of VerCors's reasoning process. Even if we do not present a transformation rule for events in this work, we outline our idea how to capture events with VerCors in the future.

4.7 Outlook: Events

Thread processes in SystemC can wait for the notification of an event, before they continue their execution. Since events are not implemented by PVL, the specification of an order of threads defined by events is not simple. However, in PVL, a thread can wait for a lock, which is very similar to the waiting for an event in SystemC. When a thread waits for a lock, the lock invariant is asserted and the lock released. If another thread acquires the lock afterwards, the lock invariant is asserted again.
Analogeously to the notifications of events, also threads can be notified of a lock, so that they can continue with their execution. VerCors only checks, whether the notifying thread holds the lock for the notification. So for our simple example SystemC design, we keep the event semantics by using the waiting for and notification of the global lock. Listing 4.17 shows the implementation in the channel class. The thread which calls `writeToBuffer(...)` suspends its execution when reaching line 14. It only continues the execution, if another thread notifies the global lock like shown in line 24. Still, the notifying thread must release the lock, otherwise no other thread can (re)acquire it.
In our general SystemC design, we only have one event involved. Real SystemC designs normally involve multiple events, which denote different global program states. The execution order of the SystemC thread processes depends on the order of event notifications, and whether a thread process has been waiting for it. Currently, these semantics of an event-driven simulation can not be formalized with PVL.
An idea to solve this problem, is the implementation of **condition variables** into PVL. They are a concurrency concept implemented by many software languages with multithreading functionality, for example Java. Condition Variables have already facilitated deductive verification of concurrent Java programs in the past [BK07]. In general, they divide the waiting threads into different sets. Each set is related to a condition variable, which denotes a different state of the program. Notifications with condition variables can target threads of a specific set instead of all threads. The challenge in the implementation of conditional variables is, that they specify global state informations, which are difficult to prove with VerCors. For example, for a concrete event, VerCors may has to verify that

```
class Channel {

  GlobalLock gLock;

  // Fields, Constructor, ...

  // Pre-/Postconditions...
  void writeToBuffer(int c) {

    loop_invariant true
      // ...
    ;
    while (n == buffer.length) {
      wait gLock;
    }

    buffer[17] = c;
    n++;
  }

  // Pre-/Postconditions...
  int returnElemNum() {
    n--;
    notify gLock;

    return n;
  }
}
```

Listing 4.17: PVL Wait and Notification of Global Lock within Channel Class.

1. a notification is made, when the event occurs,
2. the program state between release and new acquiration of the global lock with the condition variable does not change,
3. and possibly even that a thread has been be waiting for the event before it is notified.

Especially, the third assertion is related to *liveness properties*, which are not natural to prove for VerCors. This is why the concrete implementation of the conditional global lock and verification of events in VerCors is difficult and an open research topic. However, we believe, that it will be solved by future work within the SAVES project [SAVES21].
All in all, we presented transformation rules which fully preserve the non-preemptive scheduling semantics of an initial subset of SystemC designs. In addition, we identified a set of specifications that could potentially be added automatically to ease the verification process. Finally, we proposed an idea how the event mechanism can be rebuilt with PVL. What we did not show until now, is, how properties of the transformed programs can be deductively verified using VerCors. This is investigated in the next chapter.

5

Specification and Verification of Safety Properties with VerCors

In this chapter, we give an introduction how properties of SystemC designs can be formalized and verified with VerCors. In general, they can be categorized into two classes: **Safety** and **Liveness** [MK06]. Safety means that *nothing bad* will happen during the system's execution. For example, overflows, divions by zero, memory accesses out of the range, or data races are errors which violate the correct program state. Liveness expresses that *something good* will eventually happen. For instance, no livelocks can occur, and that the program has a desirable progess during its execution. The termination of a program is also part of its liveness, if it is requested as the valid final state. Moreover, there exist incorrect program states, which violate both safety and liveness of a program. For example, a deadlock is an invalid program state, and also stops its execution.
In general, liveness properties of concurrent programs are more difficult to prove than safety properties, since they involve the formalization of the scheduling semantics. VerCors does not prove total correctness, but partial correctness. It can not verify, whether a program terminates. VerCors proves all assertions under the assumption, that the considered part of the program terminates. However, it is possible to specify some properties about the program's progress, but they are very challenging to formalize in VerCors, since they need a complex encoding into invariants.
In the following, we investigate some interesting properties of integrated hardware/software systems. We focus on safety properties, because they are natural to prove with VerCors's modular verification approach. Moreover, liveness properties are easier to prove with model checking approaches than safety properties. To state that something eventually appears, it suffices to find one satisfying state. In contrast, the absence of invalid program states, which is expressed by safety, is much more difficult for model checkers, since it possibly involves very large state spaces. VerCors can abstract this state spaces very well.

5.1 Data Race Freedom and Memory Safety

VerCors's requirement to explicitly define all permissions of heap variables increases the effort of transformations from SystemC designs to PVL programs. However, the specification of permissions can be generalized and automatized with an implementation of our presented transformation rules in the future. If the resulting PVL program passes the verification, then data race freedom and memory safety are ensured by VerCors.

We illustrate the data race freedom and memory safety by a simple producer-consumer case study. Two threads, a producer and a consumer, are communicating via a shared channel implementing a FIFO. The producer writes integer values to the channel, while the consumer reads them out. The FIFO is complex due to its buffer implementation. The buffer is modeled as a *ring buffer*, which means that the next position to be written after the buffer's last position is the first position again. To count the next read and write positions, two variables `w_pos` and `r_pos` are port of the implementation. The number of elements in the buffer is denoted by an additional variable `n`.

We transformed the producer-consumer case study by following the transformation rules described in Chapter 4. The result can be found in directory `listings\prod1cons1\pvl`. In order to focus on the properties instead of transformation details, we simplified the resulting PVL program while keeping the formalization of the SystemC semantics. There is no global lock applied on the `run()` methods of the producer and consumer threads anymore. Instead a lock of the FIFO object is acquired and released in every method of the FIFO class. For this particular case study, this simplification still preserves the non-preemptive scheduling semantics, since the FIFO contains all shared heap variables. An intuition could also be, that for this case the FIFO *is* the global lock. Listing 5.1 shows the complete `Fifo` class in PVL. The consumer and producer threads are calling its `readV()` and `writeV(int c)` methods. The full source code of the case study and the following properties is available in the subdirectories under `listings\prod1cons1`.

```
1  class Fifo {
2
3    int[] buffer;
4    int n, r_pos, w_pos;
5
6  // ---------- NEEDED FOR GENERAL VERIFICATION - BEGIN -----------
7  static pure boolean validBuffer(int[] buffer, int r_pos, int w_pos)
8    = buffer != null && buffer.length != 0
9      && 0 <= w_pos && w_pos < buffer.length
10     && 0 <= r_pos && r_pos < buffer.length;
11
12 inline resource bufferPerms() = Perm(n,1) ** Perm(r_pos,1)
13   ** Perm(w_pos,1) ** Value(buffer)
14   ** validBuffer(buffer,r_pos,w_pos) ** Perm(buffer[*],1);
15
16 resource lock_invariant() = bufferPerms();
17 // ---------- NEEDED FOR GENERAL VERIFICATION - END ------------
```

```
18
19    requires 0 < BUFFERSIZE;
20    Fifo(int BUFFERSIZE) {
21      buffer = new int[BUFFERSIZE];
22      n = 0;
23      r_pos = 0;
24      w_pos = 0;
25    }
26
27    void writeV(int c) {
28      lock this;
29
30      loop_invariant bufferPerms() ** held(this);
31      while (n == buffer.length) {
32        wait(this);
33      }
34
35      buffer[w_pos] = c;
36      n = n + 1;
37      w_pos = (w_pos + 1) % buffer.length;
38
39      notify this;
40      unlock this;
41    }
42
43    int readV() {
44      lock this;
45
46      int c;
47      loop_invariant bufferPerms() ** held(this);
48      while (n == 0) {
49        wait(this);
50      }
51
52      c = buffer[r_pos];
53      n = n - 1;
54      r_pos = (r_pos + 1) % buffer.length;
55
56      notify this;
57      unlock this;
58      return c;
59    }
60  }
```

Listing 5.1: Channel Class of Producer-Consumer Case Study.

The permissions and specifications in lines 7 to 16 are necessary to verify the producer-consumer case study. The keyword `pure` in line 7 denotes that the method `validBuffer(...)` must be free of side effects. The method `validBuffer(...)` specifies null checks and value

range restrictions for the buffer, read position, and write position passed as arguments. We use pure methods for a better overview of the specifications, without the risk of accidentally modifying the program state. In lines 12 to 14, the permissions for all fields, and the valid buffer property for `buffer`, `r_pos` and `w_pos` are specified. They express the memory safety of the FIFO. The `bufferPerms()` are added to the lock invariant and to all loop invariants. By this, both data races absence, and memory safety can be guaranteed globally by VerCors. In contrast to the original SystemC design, we abstracted the exact buffer size. The constructor of the `Fifo` class (lines 19 to 25) only requires, that `BUFFERSIZE` is greater than zero. This assumption is also specified as a precondition of the `main()` method in the `Main` class, as shown in line 3 of Listing 5.2. VerCors does not need the exact size, which is a great achievement for the verification. While model checking approaches suffer from the state space explosion created by the buffer size, VerCors can even prove properties for general cases of the buffer size.

```
1  class Main {
2
3      requires 0 < BUFFERSIZE;
4      void main(int BUFFERSIZE) {
5          Fifo fifo = new Fifo(BUFFERSIZE);
6          Producer prod = new Producer(fifo);
7          Consumer cons = new Consumer(fifo);
8
9          fork prod;
10         fork cons;
11         join prod;
12         join cons;
13     }
14 }
```

Listing 5.2: Main Class of Producer-Consumer Case Study.

5.2 Variable and Buffer Overflows

In Section 4.4, we explained that VerCors has a mathematical interpretation of integers. Hence, VerCors assumes that integer values can be infinitely large. This does not hold for integrated hardware/software systems, since memory is always a limited resource. Therefore, as a common error the overflow of variables occurs. If the maximal integer value is assigned to a variable, and this variable is increased, it is wrapped and set to the lowest possible value, or an exception is thrown. To detect such errors, simple assertions can be added to the PVL code.

The producer of our producer-consumer case study writes integer values to the buffer. These values are not randomly chosen; instead they start with 0 and, after every write transaction, the next value written to the buffer is increased by 1. Listing 5.3 shows the corresponding source code of the `Producer` class in PVL.

Since VerCors does not define a maximal integer value, we added it manually to the `Producer` class (line 3). We assumed that 32 bits are used to represent an integer. For general SystemC

```
1 int produce(int c_param) {
2
3   // Property: No Variable Value Overflow (fail)
4   int INT_MAX = 2147483647;
5   assert c_param < INT_MAX;
6
7   c_param = c_param + 1;
8   return c_param;
9 }
```

Listing 5.3: `produce()` method of Producer of Producer-Consumer Case Study.

designs our idea is to add the constants for minimal and maximal values to the global lock class. In this case study, we specified an assertion of the value of `c_param` in line 5. If `c_param` is smaller than the maximal integer value, its subsequent incrementation is safe. In our case study, this assertion *does not pass*, i. e., there is a risk of an arithmetical overflow for this value during the program's execution.
Another possible overflow in the producer-consumer case study is related to the buffer capacity. The variable `n` denotes the number of elements in the buffer, but its range is not restricted yet. To verify, that n never exceeds the buffer capacity, we specify this behavior in the FIFO class by an additional inline resource:

```
inline resource noBufferOverflow() = (0 <= n && n <= buffer.length);
```

The resource `noBufferOverflow()` is added to the lock and all loop invariants. It passes for the producer-consumer case study. However, until now, we have not considered that the FIFO is realized by a ring buffer. Already read elements will be overwritten again, so the more interesting safety property is whether an unread element is eventually overwritten. This property is specific to the functionality of the producer-consumer case study.

5.3 Written Buffer Data is Eventually Read

The program state remains safe, if an unread element of the buffer is never overwritten. The stronger property, which is even more interesting, is, that all elements are eventually received in the future. Since this also expresses a liveness property of the program, it is very challenging to prove it with VerCors. At the same time, it is not easy to be proved with model checking approaches as well, but suffers from other parts of the design. Herber, Fellmuth, and Glesner [HFG08] investigated a producer-consumer case study with two producers and one consumer, and verified whether the number of buffer elements `n` is kept in range. They checked their FIFO model with multiple buffer sizes from 10 up to 1000. The verification time of the model checker Uppaal increased exponentially and took approximately 3 hours for a buffer size of 1000. If even a much simpler property lets the verification time of model checking scale poorly, the verification time of a more complex property like "written data is eventually read" with large buffer sizes will behave even worse.

Currently, we can not prove that all written elements will eventually be read with VerCors, because this is an assertion of the global progress of the program state. Instead, we can prove a related and slightly weaker property: All values, which have been written to the buffer, are either still waiting in the buffer, or have been read. It does not state, whether the waiting elements will be read eventually, but that they are not lost. Hence, no unsafe program state can be reached. For simplicity in the following sections, we will refer to the weaker property as the "written data is read" property.

In order to prove the all written elements are read, we need to reason about every newly written or read element. For this, we have to save them outside of the buffer, since the ring buffer overwrites its positions with new elements. Without extra saving of the elements, we have no knowledge about their state, since VerCors can not reason about a past or future state of the buffer without extra effort.

Therefore, we add *ghost code* to the FIFO source code. Ghost code are statements, which are only used for specification purposes, and must not change the program state. In Section 3.3.3, we introduced the axiomatic data type sequence, which is also some sort of ghost code. In contrast to arrays, sequences have the advantage, that they are immutable and easier to use in specifications. It is not necesary to have permissions for their elements, or to state specifications of their lengths. At the same time, a field holding a sequence can be reassigned to a new extended sequence. Consequently, they fit very well as a storage recording all ever written or read elements.

Listing 5.4 shows the `Fifo` class annotated with ghost code, highlighted by yellow background. We add two sequences `sent` and `rcvd` as fields to the `Fifo` class (Lines 4f). The only necessary permissions **Perm**`(sent, 1)` and **Perm**`(rcvd, 1)` are added to the lock and loop invariants (Lines 17, 37, 58 and 81). Furthermore, we synchronize the `sent` and `rcvd` sequences with the buffer within the `writeV` and `readV` methods. Every time, a new element is written, it is also added to the `sent` sequence (Line 66). Accordingly, every read element is appended to the `rcvd` sequence (Line 89).

```
1  class Fifo {
2    // SPECIFICATION ONLY - BEGIN (Property: Sent Data is Received)
3    // ------------- DATA OBJECTS -------------------------------------
4    seq<int> sent;
5    seq<int> rcvd;
6    // ------------- DATA OBJECTS -------------------------------------
7
8    // ------------- HELPER FUNCTIONS ------------------------------
9    requires 0 <= value && 0 < m && (value - m) < m;
10   ensures (value < m) ==> \result == value;
11   ensures (value >= m) ==> \result == (value - m);
12   static pure int wrap(int value, int m)
13     = (value < m ? value : value - m);
14   // ------------- HELPER FUNCTIONS ------------------------------
15
16   // ------------- RESOURCES AND PREDICATES (Lemmas) ---------------
```

```
17    inline resource seqPerms() = Perm(sent, 1) ** Perm(rcvd, 1);
18    // ...
19    // ------------- RESOURCES AND PREDICATES (Lemmas) ---------------
20    // SPECIFICATION ONLY - END (Property: Sent Data is Received)
21
22    int[] buffer;
23    int n, r_pos, w_pos;
24
25    // ------------- NEEDED FOR GENERAL VERIFICATION - BEGIN ---------
26    static pure boolean validBuffer(int[] buffer, int r_pos, int w_pos)
27    = buffer != null && buffer.length != 0
28      && 0 <= w_pos && w_pos < buffer.length
29      && 0 <= r_pos && r_pos < buffer.length;
30
31    inline resource bufferPerms() = Perm(n,1) ** Perm(r_pos,1)
32      ** Perm(w_pos,1) ** Value(buffer)
33      ** validBuffer(buffer,r_pos,w_pos) ** Perm(buffer[*],1);
34
35    resource lock_invariant() = true
36      ** bufferPerms()
37      ** seqPerms()
38    // ** lemmas
39    ;  // from 2nd line: ghost code
40    // ------------- NEEDED FOR GENERAL VERIFICATION - END -----------
41
42
43    requires 0 < BUFFERSIZE;
44    Fifo(int BUFFERSIZE) {
45      buffer = new int[BUFFERSIZE];
46      n = 0;
47      r_pos = 0;
48      w_pos = 0;
49      sent = seq<int> {};   // ghost code
50      rcvd = seq<int> {};   // ghost code
51    }
52
53    void writeV(int c) {
54      lock this;
55
56      loop_invariant bufferPerms()
57      ** held(this)
58      ** seqPerms()      // ghost code
59      // ** lemmas
60      ;
61      while (n == buffer.length) {
62        wait(this);
63      }
64
65      buffer[w_pos] = c;
```

```
66      sent = sent ++ c;      // ghost code
67      n = n + 1;
68      w_pos = wrap(w_pos + 1, buffer.length);
69
70      notify this;
71      unlock this;
72    }
73
74
75    int readV() {
76      lock this;
77
78      int c;
79      loop_invariant bufferPerms()
80      ** held(this)
81      ** seqPerms()      // ghost code
82      // ** lemmas
83      ;
84      while (n == 0) {
85        wait(this);
86      }
87
88      c = buffer[r_pos];
89      rcvd = rcvd ++ c;      // ghost code
90      n = n - 1;
91      r_pos = wrap(r_pos + 1, buffer.length);
92
93      notify this;
94      unlock this;
95      return c;
96    }
97
98  }
```

Listing 5.4: `Fifo` Class of Producer-Consumer Case Study extended by Ghost Code.

The buffer should contain the elements, which are written, but not yet read. Since `w_pos` points to the next position which should be written, and `r_pos` to the next element to be read, the elements of interest are exactly located between the counters. Therefore, to reason about written, but not read elements, we must relate `r_pos`, `w_pos` and `n` in the specifications.

For this, we have to replace the modulo operations for the buffer wrap (Lines 37 and 54 in Listing 5.1). Silicon, the verification backend tool used by VerCors, can not handle modulo operations very well, therefore, we rewrite them to a more simple call of a helper function `wrap()`, which is defined in Lines 9 to 13 in Listing 5.4. The adjustments are also highlighted for a more convenient distinction.

The lines 16 and 19 of Listing 5.4 show, that we have to define some lemmas. The property, that all sent elements are received, must be differently formulated depending on how `r_pos`, `w_pos` and `n` are related to each other. For a better overview, we divide these different cases, and specify them in seperate lemmas. All of them are added to the lock and loop invariants. Listing 5.5 shows all necessary lemmas.

```
1  // -------- RESOURCES AND PREDICATES (Lemmas) ------------------
2  inline resource validNumberOfElems()
3    = (0 <= n && n <= buffer.length);
4
5  inline resource sentLengthEqRcvdLengthPlusNLemma()
6    = (|sent| == (|rcvd| + n));
7
8  // Conclusions by Relations of r_pos and w_pos
9  inline resource readLessWritePosLemma()
10   = (r_pos < w_pos ==> (n == w_pos - r_pos));
11
12 inline resource readGreaterWritePosLemma()
13   = (r_pos > w_pos ==> (n == buffer.length + w_pos - r_pos));
14
15 inline resource readEqWritePosLemma()
16   = ((r_pos == w_pos) ==> ((n == 0) || (n == buffer.length)));
17
18 // Conclusions by Value of n
19 inline resource zeroElemsLemma()
20   = (n == 0) ==> (r_pos == w_pos);
21
22 inline resource maxNumberOfElemsLemma()
23   = (n == buffer.length) ==> (r_pos == w_pos);
24
25 // Conclusions by Positive Values of w_pos and r_pos
26 inline resource writeIncrSentLengthLemma()
27   = (w_pos > 0) ==> (|sent| > 0);
28
29 inline resource readIncrRcvdLengthLemma()
30   = (r_pos > 0) ==> (|rcvd| > 0);
31
32 // Synchronization of Buffer and Sent Sequence Lemmas
33 inline resource notRcvdSentElemsLemma0()
34   = (w_pos > 0) ==> (buffer[w_pos-1] == sent[|sent|-1]);
35
36 inline resource notRcvdSentElemsLemma1()
37   = (w_pos == 0 && |sent| > 0)
38     ==> (buffer[buffer.length-1] == sent[|sent|-1]);
39
40 // Conclusions by Number of Elements n
41 inline resource existNumberOfElemsLemma()
42   = (0 < n && n < buffer.length) ==> (r_pos != w_pos);
43
```

```
// Sent Data is Received depending on the Different Cases
inline resource sentDataIsReceived() = true
  && (((r_pos == w_pos) && n == buffer.length)
    ==>  (\forall int i; r_pos <= i && i < buffer.length;
        buffer[i] == sent[|sent| - n - r_pos + i])
      )
  && (((r_pos == w_pos) && n == buffer.length)
    ==> (\forall int i; 0 <= i && i < r_pos ;
        buffer[i] == sent[|sent| - n + i])
      )

  && (((r_pos == w_pos) && n == 0)
    ==> |sent| == |rcvd|)
  && (((r_pos == w_pos) && n == 0)
    ==> (\forall int i; 0 <= i && i < |sent|;
        sent[i] == rcvd[i])
        )

  && ((r_pos < w_pos)
    ==> (\forall int i; r_pos <= i && i < n + r_pos ;
        buffer[i] == sent[|sent| - n - r_pos + i])
      )
  && ((r_pos > w_pos)
    ==> (\forall int i; r_pos <= i && i < buffer.length;
      buffer[i] == sent[|sent| - n - r_pos + i])
    )
  && ((r_pos > w_pos)
    ==> (\forall int i; 0 <= i && i < w_pos;
        buffer[i] == sent[|sent| - n + i])
      )

  && r_pos != w_pos ==> n > 0
;
// -------- RESOURCES AND PREDICATES (Lemmas) ------------------
```

Listing 5.5: Lemmas for "written data is read" Property of Producer-Consumer Case Study.

We explain what the lemmas express and why they are needed.

1. **validNumberOfElems():** We already introduced this assertion as buffer overflow absence. In this context, it also has an additional meaning: The number of written, but unread elements must always be between zero and the maximal buffer size.
2. **sentLengthEqRcvdLengthPlusNLemma():** The number of written elements must be equal to the number of read elements summed up with the number of written, but not read elements.
3. **readLessWritePosLemma():** If the read position is smaller than the write position, the elements of the buffer are not wrapped around them, but are located between the

read and write position.

4. **readGreaterWritePosLemma():** If the write position ist greater than the read position, the elements are wrapped around the buffer. In detail, the written, but unread elements are located from the read position up to the buffer length, and also from the buffer start to the read position.
5. **readEqWritePosLemma():** If the read and write positions are equal, either the buffer is empty or it is full.
6. **zeroElemsLemma():** If no elements are in the buffer, the read and write position must be equal.
7. **maxNumberOfElemsLemma():** If the buffer is full, the read position equals the write position.
8. **writeIncrSentLengthLemma():** If the write position is greater than zero, at least one element must have been written to the buffer. Therefore the size of the `sent` sequence must be greater than zero.
9. **readIncrRcvdLengthLemma():** If the read position is greater than zero, at least one element must have been read. Therefore the size of the `rcvd` sequence must be greater than zero.
10. **notRcvdSentElemsLemma0():** If the write position is greater than zero, then at least one element has been written to the buffer. It follows, that the last written element in the buffer is equal to the last element of the `sent` sequence.
11. **notRcvdSentElemsLemma1():** If the write position is equal to zero, but still an element has been written, then a wrap of the write position has happened, and the last written element is located at the end of the buffer. It is still equal to the last element of the `sent` sequence.
12. **existNumberOfElemsLemma():** If written, but unread elements exist in the buffer, then the read and write positions can not be equal.
13. **writtenDataIsRead():** This is not a lemma like the other ones, but rather the main assertion of the property. Seperated into different cases, we formalize that all elements of the `sent` sequence are either contained in the `rcvd` sequence or in the buffer. Since this holds always and globally, we can conclude, that no element is written, but overwritten before it is read.

The lemmas are necessary to give VerCors some extra knowledge to perform the verification. If one of the lemmas is removed, VerCors can not prove the “written data is read” property anymore. For our producer-consumer case study, we could verify the “written data is read” property. There are multiple aspects of the FIFO case study, which makes the “written data is read” property so complex to prove:

1. The arithmetical relations of the read position, write position and the number of elements,
2. the wrapping property of the ring buffer,
3. the abstracted buffer size, which is not explicitly defined except for being positive.

During our research, we experienced, that even the order of the stated lemmas decides on a failing or a passing verification. If the simple assertion in Line 75 of Listing 5.5 is moved upwards and exchanged with the assertion in Line 70, the verification fails. We assume that this is related to the heuristics and randomization used by Silicon in the backend. Even though a change of the order of boolean expressions within a conjunction does not change the logical meaning of the formula, it triggers different approaches in VerCor's technical backend. The relevance of the order increased the difficulty to correctly formalize properties with VerCors, but still, it has been possible to prove an important functional property. In the future, we plan to investigate more detailed how the order of boolean formulas is related to the verification success.

6 Conclusion and Outlook

In this chapter, we sum up our contributions to the deductive verification of integrated hardware/software systems. For this, we discuss the results of the previous chapters and relate them to the research objective we stated in the beginning of this thesis. Afterwards, we outline possible improvements and extensions of our research approach, and present proposals for future work.

6.1 Results and Contributions

All approaches to verify integrated hardware/software systems in the past shared the problem of limited scalability. In contrast, deductive verification offers the advantage, that is must not investigate the whole state space of a program in order to prove properties of it. In particular, the VerCors Verifier performs a static analysis of the input program, thereby the verification is done without any execution at all. Its modular approach enables the abstraction of program components, like the buffer size of the producer-consumer case study presented in Chapter 5.

We introduced SystemC as a representative modeling language for the design of integrated hardware/software systems. Its key feature is an event-driven simulation, whose scheduling is performed cooperatively. However, the semantics of the SystemC language are only informally defined. To verify them, we need to formalize them first. Therefore, we investigated how to manually transform SystemC Designs to annotated PVL programs.

In Chapter 4, we introduced multiple contributions to a semantics-preserving transformation from SystemC Designs to PVL programs. First, we presented transformation rules for an initial subset of SystemC constructs. Currently, these transformation rules must be considered manually, but they can be easily automated by an implementation in the future. A particular challenge for the development of the transformation rules was the preservation of the cooperative scheduling schemantics. We achieved this objective by developing a global lock, which controls the permissions for all heap variables of the program. While

the specifications of the result program must be added manually at this time, and therefore increase the effort for the designer, we already outlined a set of specifications, that could potentially be added automatically to ease the verification. Furthermore, we outlined our ideas how to implement the event mechanism of the SystemC scheduler into PVL. We demonstrated our transformation on a small SystemC design and verified it with VerCors. Besides the transformation, we also showed how interesting properties of SystemC designs can be formalized in VerCors's specification language in Chapter 5. The absence of data races, and memory safety are built-in features of VerCors. Also the the absence of arithmetical or buffer overflows is easy to specify with VerCors. The last presented property, that all elements written into a ring a buffer by a producer, are either still in the buffer or have been read by the a consumer, was much more complex to prove. One of the difficulties has been, that the property expresses an assertion of the global program state and its progress. This is not natural for VerCors to verify. To overcome the challenge, we had to add several lemmas to give VerCors extra knowledge about the program state.
In Section 1.2, we defined as our research goal to investigate deductive verification techniques for integrated harware/software systems. We stated, that our approach should be semantics-preserving and scalable with respect to the number and size of the system's components. After working several months on deductive verification of SystemC designs, we draw the conclusion, that the development of a semantics-preserving transformation for general SystemC designs without restrictions of the supported language set is very ambitious, but achievable in the future. We think that this thesis has contributed in the following ways:

1. We presented informal **transformation rules** which precisely capture an initial subset of SystemC designs.
2. We developed a **global lock mechanism** which preserves the cooperative scheduling semantics during the transformation.
3. We could **deduce an automatable set of specifications from the transformed program** to ease the verification.

Even if the presented small SystemC example design has been transformed manually for this thesis, our approach is extensible to the automation of multiple parts of the transformation process. We describe the improvements in Section 6.2.
Our research objective has also been to identify interesting properties to verifiy, and to evaluate their verification in appropriate case studies. Besides the easy specifiation and verification of overflow properties, we also have proven a complex property of a Producer Consumer design sharing a FIFO implemented by a ring buffer. We could not prove the stronger property, that all written elements are read eventually, but have proven the slightly weaker property, that all written elements are either read or still in the buffer. The investigation of these properties gave us an intuition, which sort of properties are more difficult to prove with VerCors and why. Furthermore, this case study is representative for

several SystemC designs, so we think the gained knowledge will improve the formulation of further and even more complex properties in the future.

6.2 Future Work

We outlined the results of this thesis, now we present how the work can be enhanced in future. Our approach of deductive verification of integrated hardware/software systems is designed to be extendable. Therefore, it can be improved in multiple ways.
First of all, we have ideas to improve the transformation process. The semantics-preserving transformation can become more easy to use, if it is automated by an implementation of the presented transformation rules. Hence, we see this as a first, but important step for future work. Furthermore, we plan to extend the supported subset of the SystemC language. Construcs like Method Processes, Static Sensitivity, and more have been left out in this work for simplicity, but their support would increase the usability of our transformation and make it more complete. Especially, the support of SystemC's event mechanism is an important objective we would like to implement in future. Its realization will increase the set of supported SystemC designs significantly.
For some improvements, enhanced functionality of the VerCors Verifier is necessary. We mentioned that the VerCors team is already working on a floating point data type, which would be interesting for the transformation of SystemC's floating point based data types as well. Besides, SystemC offers bit-precise data types, whose support by VerCors would be very interesting. Currently, also the implementation of inheritance into VerCors gets investigated and may enable the support of more hierarchical SystemC designs. In general, we see every new feature of the VerCors Verifier as a possiblity to improve our transformation rules and to support more interesting SystemC designs.
Furthermore, we formalized a small selection of safety properties for SystemC designs and verified them for the producer-consumer case study. Based on the knowledge we gained by this research work, we would like to verify more properties of different case studies with VerCors. For instance, Anti-lock Braking Systems and Anti-Slip Regulation Systems are very interesting SystemC designs and have properties to prove, which are similar to the "written elements are read" property of our producer-consumer case study. We also consider an experimental comparison of the logical backends used by VerCors. In this thesis, we used Silicon, which is based on sound symbolic execution. Another tool of the Viper architecture is Carbon, which is a verification-condition-generation-based verifier. It is possible, that some properties are easier to verify with Carbon instead of Silicon.
In summary, we contributed to the deductive verification of integrated hardware/software systems by multiple achivements presented in this thesis. Furthermore, we proposed ideas how to improve and extend the results. We conclude, that the approach to use deductive verification techniques for integrated hardware/software systems is promising, and should be furtherly pursued in future.

A
Appendix

Getting Started	This menu point shows a constantly expanded wiki/tutorial. It covers topics like an installation guide, PVL syntax, the specification language syntax, permissions, axiomatic data types, atomics and locks, resources and predicates, and how to understand error messages produced by VerCors.
Showcases	Here, a lot of already verified examples are shown. They can be filtered by, inter alia, the example title, its verification features, and implementation language.
Publications	A complete list of all VerCors-related publications. The first publication is from 2008—VerCors's founding year—up to publications which have been just recently published.
Try VerCors Online	Instead of downloading an executable version of VerCors, it is possible to try out very simple examples with the online verification user interface.

Table A.1: VerCors Website Guide.

A.1 SystemC Design Example for Transformation

The full source code is also available as compilable files under `listings\transformation\sc`.

```
#include <systemc.h>
#include "channel.h"
#include "moduleA.h"
#include "moduleB.h"

int sc_main(int argc, char* argv[]) {
    // Elaboration Phase
    channel c_inst("Channel");

    moduleA mA_inst("ModuleA");
    mA_inst.channelA(c_inst);

    moduleB mB_inst("ModuleB");
    mB_inst.channelB(c_inst);

    // Simulation of Processes
    sc_start();

    // No Further Post Processing
    return 0;   // Simulation Success
}
```

Listing A.1: SystemC Main Function (`main.cpp`).

```
#include <systemc.h>

SC_MODULE(moduleA) {
    // Ports connected to Channels
    sc_port<channel_if> channelA;

    // Constructor
    SC_CTOR(moduleA) {
        // Processes (No Static Sensitivity)
        SC_THREAD(thread);
    }

    // Interruptable Process
    void thread(void) {
        while(true) {
            // ...
        }
    }
};
```

Listing A.2: SystemC Module A (`moduleA.h`).

```
#include <systemc.h>

SC_MODULE(moduleB) {
    // Ports connected to Channels
    sc_port<channel_if> channelB;

    // Member Data Instances
    int numberOfPorts;

    // Constructor
    SC_CTOR(moduleB) {
        // Processes (No Static Sensitivity)
        SC_THREAD(thread1);
        SC_THREAD(thread2);

        // Other Initialization
        numberOfPorts = 2;
    }

    // Immediate Computations
    int returnZero(bool b) {
        if(b) {
            return 0;
        } else {
            return -1;
        }
    }

    // Interruptable Process 1
    void thread1(void) {
        while(true) {
            returnZero(true);
            channelB->writeToBuffer(42);
            // ...
        }
    }

    // Interruptable Process 2
    void thread2(void) {
        while(true) {
            channelB->writeToBuffer(3);
            // ...
        }
    }
};
```

Listing A.3: SystemC Module B (`moduleB.h`).

```
#include <systemc.h>
#define BUFFERSIZE 42

class channel_if : virtual public sc_interface {
public:
    virtual void writeToBuffer(int c) = 0;
    virtual int returnElemNum(void) = 0;
};

// Channel Class implements Channel Interface
class channel : public sc_channel,
                public channel_if {

    // Internal Data
    private:
        int n;
        int buffer[BUFFERSIZE];

    // Public Events and Functions
    public:
        sc_event decrElemNumEvent;

        SC_CTOR(channel){
            n = 0;
        }

        void writeToBuffer(int c) {
            while (n == BUFFERSIZE) {
                 wait(decrElemNumEvent);
            }

            buffer[17] = c;
            n++;
        }

        int returnElemNum(void) {
            n--;
            notify(decrElemNumEvent);

            return n;
        }
};
```

Listing A.4: SystemC Channel (`channel.h`).

A.2 Transformation Rewritings Rules

The following tables show an overview over the translation rules for standard operators and data types. More explanations, and the mapping of further language constructs are given in Chapter 4.

SystemC	PVL	Remarks
`&&`	`&&`	
`||`	`||`	
`!`	`!`	
`!=`	`!=`	
`==`	`==`	
`<`	`<`	
`<=`	`<=`	
`>`	`>`	
`>=`	`>=`	
`+`	`+`	
`-`	`-`	
`*`	`*`	
`/`	`/`	
`++`	`++`	
`--`	`--`	
`%`	`%`	Possibly replaced for specifications.

Table A.2: Supported Subset of Standard Operators.

SystemC	PVL	Remarks
`int`	`int`	Particular handling of numerical range required.
`bool`	`boolean`	
`void`	`void`	If as return type.
`void`		If as argument type, then *removed*.
`T array[i];`	`T[] array;`	length `i` only necessary in initalization

Table A.3: Transformation Rewriting Rules for Data Types.

A.3 Resulting PVL Program after Transformation

The full source code is also available as verifiable files under `listings\transformation\pvl`. It can be verified by calling (within one command line call):

```
vercors --silicon Main.pvl GlobalLock.pvl ModuleA.pvl ModuleBState.pvl
        ModuleBThread1.pvl ModuleBThread2.pvl Channel.pvl
```

```
class Main {
  void main() {
    GlobalLock gLock = new GlobalLock();
    gLock.simulation();
  }
}
```

Listing A.5: PVL Main (`Main.pvl`).

```
class GlobalLock {

  // Fields for Channels, Threads, and Shared Module Data
  Channel channel_inst;
  ModuleA moduleA_inst;
  ModuleBState moduleB_inst_State;
  ModuleBThread1 moduleB_inst_Thread1;
  ModuleBThread2 moduleB_inst_Thread2;

  // Lock Invariant asserted at lock and unlock Statements
  resource lock_invariant() = true
    // Read Permissions for all Channel, State and Thread Classes
    ** Perm(channel_inst, read)
    ** Perm(moduleA_inst, read)
    ** Perm(moduleB_inst_State, read)
    ** Perm(moduleB_inst_Thread1, read)
    ** Perm(moduleB_inst_Thread2, read)

    // Read Permissions & Equality of the Shared Global Lock
    ** Perm(channel_inst.gLock, read)
    ** Perm(moduleA_inst.gLock, read)
    ** Perm(moduleB_inst_State.gLock, read)
    ** Perm(moduleB_inst_Thread1.gLock, read)
    ** Perm(moduleB_inst_Thread2.gLock, read)
    ** this == channel_inst.gLock
    ** this == moduleA_inst.gLock
    ** this == moduleB_inst_State.gLock
    ** this == moduleB_inst_Thread1.gLock
    ** this == moduleB_inst_Thread2.gLock

    // Read Permissions & Equality of Shared Module States
    ** Perm(moduleB_inst_Thread1.moduleB_inst_State, read)
    ** Perm(moduleB_inst_Thread2.moduleB_inst_State, read)
```

```
34      ** moduleB_inst_Thread1.moduleB_inst_State == moduleB_inst_State
35      ** moduleB_inst_Thread2.moduleB_inst_State == moduleB_inst_State
36
37      // Read Permissions & Equality of Shared Channels
38      ** Perm(moduleA_inst.channelA, read)
39      ** Perm(moduleB_inst_State.channelB, read)
40      ** channel_inst == moduleA_inst.channelA
41      ** channel_inst == moduleB_inst_State.channelB
42
43      // Permissions, Null and Length Checks of further Heap Variables
44      ** Perm(moduleB_inst_State.numberOfPorts, write)
45      ** Perm(channel_inst.n, write)
46      ** Perm(channel_inst.buffer, read)
47      ** channel_inst.buffer != null
48      ** channel_inst.buffer.length == 42
49      ** Perm(channel_inst.buffer[*],write)
50    ;
51
52
53    // Read Permissions and Null Check for Channels, Thread, State Classes
54    ensures Perm(channel_inst, read);
55    ensures Perm(moduleA_inst, read);
56    ensures Perm(moduleB_inst_State, read);
57    ensures Perm(moduleB_inst_Thread1, read);
58    ensures Perm(moduleB_inst_Thread2, read);
59    ensures channel_inst != null;
60    ensures moduleA_inst != null;
61    ensures moduleB_inst_State != null;
62    ensures moduleB_inst_Thread1 != null;
63    ensures moduleB_inst_Thread2 != null;
64    // Read Permissions and Null Check for Global Lock
65    ensures Perm(channel_inst.gLock, read);
66    ensures Perm(moduleA_inst.gLock, read);
67    ensures Perm(moduleB_inst_State.gLock, read);
68    ensures Perm(moduleB_inst_Thread1.gLock, read);
69    ensures Perm(moduleB_inst_Thread2.gLock, read);
70    ensures channel_inst.gLock != null;
71    ensures moduleA_inst.gLock != null;
72    ensures moduleB_inst_State.gLock != null;
73    ensures moduleB_inst_Thread1.gLock != null;
74    ensures moduleB_inst_Thread2.gLock != null;
75    // Equality of the Shared Global Lock
76    ensures channel_inst.gLock == this;
77    ensures moduleA_inst.gLock == this;
78    ensures moduleB_inst_State.gLock == this;
79    ensures moduleB_inst_Thread1.gLock == this;
80    ensures moduleB_inst_Thread2.gLock == this;
81    // Status of ModuleB's Thread1 and Thread2, and ModuleA's Thread
82    ensures idle(moduleB_inst_Thread1);
```

```
83    ensures idle(moduleB_inst_Thread2);
84    ensures idle(moduleA_inst);
85    GlobalLock () {
86      channel_inst = new Channel(this);
87      moduleB_inst_State = new ModuleBState(this, channel_inst);
88      moduleB_inst_Thread1 = new ModuleBThread1(this, moduleB_inst_State);
89      moduleB_inst_Thread2 = new ModuleBThread2(this, moduleB_inst_State);
90      moduleA_inst = new ModuleA(this, channel_inst);
91    }
92
93    // Read Permissions and Null Check for Channels, Thread, State Classes
94    context Perm(channel_inst, read);
95    context Perm(moduleA_inst, read);
96    context Perm(moduleB_inst_State, read);
97    context Perm(moduleB_inst_Thread1, read);
98    context Perm(moduleB_inst_Thread2, read);
99    context channel_inst != null;
100   context moduleA_inst != null;
101   context moduleB_inst_State != null;
102   context moduleB_inst_Thread1 != null;
103   context moduleB_inst_Thread2 != null;
104   // Read Permissions and Null Check for Global Lock
105   context Perm(channel_inst.gLock, read);
106   context Perm(moduleA_inst.gLock, read);
107   context Perm(moduleB_inst_State.gLock, read);
108   context Perm(moduleB_inst_Thread1.gLock, read);
109   context Perm(moduleB_inst_Thread2.gLock, read);
110   context channel_inst.gLock != null;
111   context moduleA_inst.gLock != null;
112   context moduleB_inst_State.gLock != null;
113   context moduleB_inst_Thread1.gLock != null;
114   context moduleB_inst_Thread2.gLock != null;
115   // Equality of the Shared Global Lock
116   context channel_inst.gLock == this;
117   context moduleA_inst.gLock == this;
118   context moduleB_inst_State.gLock == this;
119   context moduleB_inst_Thread1.gLock == this;
120   context moduleB_inst_Thread2.gLock == this;
121   // Status of ModuleB's Thread1 and Thread2, and ModuleA's Thread
122   context idle(moduleB_inst_Thread1);
123   context idle(moduleB_inst_Thread2);
124   context idle(moduleA_inst);
125   void simulation() {
126     fork moduleB_inst_Thread1;
127     fork moduleB_inst_Thread2;
128     fork moduleA_inst;
129
130     join moduleB_inst_Thread1;
131     join moduleB_inst_Thread2;
```

```
132     join moduleA_inst;
133   }
134 }
```

Listing A.6: PVL Global Lock (`GlobalLock.pvl`).

```
1  class ModuleA {
2
3    GlobalLock gLock;
4
5    Channel channelA;
6
7    // NotNull Check of Arguments
8    requires globalLock != null;
9    requires channel != null;
10   // Global Lock: Permissions and Value Checks
11   ensures Perm(gLock, read);
12   ensures gLock != null;
13   ensures gLock == globalLock;
14   // Channel: Permissions and Value Checks
15   ensures Perm(channelA, read);
16   ensures channelA != null;
17   ensures channelA == channel;
18   ModuleA (GlobalLock globalLock, Channel channel) {
19     gLock = globalLock;
20     channelA = channel;
21   }
22
23   context Perm(gLock, read) ** gLock != null;
24   context Perm(gLock.moduleA_inst, read);
25   context gLock.moduleA_inst == this;
26   void run () {
27     lock gLock;
28
29     loop_invariant true
30       // Permissions and Ownership of Global Lock
31       ** Perm(gLock, read)
32       ** gLock != null
33       ** held(gLock)
34
35        // Read Permissions for all Channel, State and Thread Classes
36       ** Perm(gLock.channel_inst, read)
37       ** Perm(gLock.moduleA_inst, read)
38       ** Perm(gLock.moduleB_inst_State, read)
39       ** Perm(gLock.moduleB_inst_Thread1, read)
40       ** Perm(gLock.moduleB_inst_Thread2, read)
41
42       // Equality of Global Lock's Thread1 and this
43       ** gLock.moduleA_inst == this
44
```

```
45        // Read Permissions & Equality of the Shared Global Lock
46        ** Perm(gLock.channel_inst.gLock, read)
47        ** Perm(gLock.moduleA_inst.gLock, read)
48        ** Perm(gLock.moduleB_inst_State.gLock, read)
49        ** Perm(gLock.moduleB_inst_Thread1.gLock, read)
50        ** Perm(gLock.moduleB_inst_Thread2.gLock, read)
51        ** gLock == gLock.channel_inst.gLock
52        ** gLock == gLock.moduleA_inst.gLock
53        ** gLock == gLock.moduleB_inst_State.gLock
54        ** gLock == gLock.moduleB_inst_Thread1.gLock
55        ** gLock == gLock.moduleB_inst_Thread2.gLock
56
57        // Read Permissions & Equality of Shared Module States
58        ** Perm(gLock.moduleB_inst_State, read)
59        ** Perm(gLock.moduleB_inst_Thread1.moduleB_inst_State, read)
60        ** Perm(gLock.moduleB_inst_Thread2.moduleB_inst_State, read)
61        ** gLock.moduleB_inst_Thread1.moduleB_inst_State
62          == gLock.moduleB_inst_State
63        ** gLock.moduleB_inst_Thread2.moduleB_inst_State
64          == gLock.moduleB_inst_State
65
66        // Read Permissions & Equality of Shared Channels
67        ** Perm(gLock.channel_inst, read)
68        ** Perm(gLock.moduleA_inst.channelA, read)
69        ** Perm(gLock.moduleB_inst_State.channelB, read)
70        ** gLock.channel_inst == gLock.moduleA_inst.channelA
71        ** gLock.channel_inst == gLock.moduleB_inst_State.channelB
72
73        // Permissions, Null and Length Checks of further Heap Variables
74        ** Perm(gLock.moduleB_inst_State.numberOfPorts, write)
75        ** Perm(gLock.channel_inst.n, write)
76        ** Perm(gLock.channel_inst.buffer, read)
77        ** gLock.channel_inst.buffer != null
78        ** gLock.channel_inst.buffer.length == 42
79        ** Perm(gLock.channel_inst.buffer[*],write)
80        ;
81      while(true) {
82        // ...
83      }
84
85      unlock gLock;
86    }
87  }
```

Listing A.7: PVL Module A (`ModuleA.pvl`).

```
1   class ModuleBState {
2
3     GlobalLock gLock;
4
```

```
  Channel channelB;

  // Internal Data
  int numberOfPorts;

  // NotNull Check of Arguments
  requires globalLock != null;
  requires channel != null;
  // Global Lock: Permissions and Value Checks
  ensures Perm(gLock, read);
  ensures gLock != null;
  ensures gLock == globalLock;
  // Channel: Permissions and Value Checks
  ensures Perm(channelB, read);
  ensures channelB != null;
  ensures channelB == channel;
  // numberOfPorts: Permissions and Value Checks
  ensures Perm(numberOfPorts, write);
  ensures numberOfPorts == 1;
  ModuleBState(GlobalLock globalLock, Channel channel) {
    gLock = globalLock;
    channelB = channel;
    numberOfPorts = 1;
  }

  // Immediate Computations
  int returnZero(boolean b) {
    if (b) {
      return 0;
    } else {
      return -1;}
  }
}
```

Listing A.8: PVL Shared Module B State (`ModuleBState.pvl`).

```
class ModuleBThread1 {

  GlobalLock gLock;

  ModuleBState moduleB_inst_State;

  // NotNull Check of Arguments
  requires globalLock != null;
  requires moduleBState != null;
  // Global Lock: Permissions and Value Checks
  ensures Perm(gLock, read);
  ensures gLock != null;
  ensures gLock == globalLock;
  // Shared ModuleBState: Permissions and Value Checks
```

```
15    ensures Perm(moduleB_inst_State, read);
16    ensures moduleB_inst_State != null;
17    ensures moduleB_inst_State == moduleBState;
18    ModuleBThread1(GlobalLock globalLock, ModuleBState moduleBState) {
19      gLock = globalLock;
20      moduleB_inst_State = moduleBState;
21    }
22
23
24    context Perm(gLock, read);
25    context gLock != null;
26    context Perm(gLock.moduleB_inst_Thread1, read);
27    context gLock.moduleB_inst_Thread1 == this;
28    void run () {
29      lock gLock;
30
31      loop_invariant true
32        // Permissions and Ownership of Global Lock
33        ** Perm(gLock, read)
34        ** gLock != null
35        ** held(gLock)
36
37         // Read Permissions for all Channel, State and Thread Classes
38        ** Perm(gLock.channel_inst, read)
39        ** Perm(gLock.moduleA_inst, read)
40        ** Perm(gLock.moduleB_inst_State, read)
41        ** Perm(gLock.moduleB_inst_Thread1, read)
42        ** Perm(gLock.moduleB_inst_Thread2, read)
43
44        // Equality of Global Lock's Thread1 and this
45        ** gLock.moduleB_inst_Thread1 == this
46
47        // Read Permissions & Equality of the Shared Global Lock
48        ** Perm(gLock.channel_inst.gLock, read)
49        ** Perm(gLock.moduleA_inst.gLock, read)
50        ** Perm(gLock.moduleB_inst_State.gLock, read)
51        ** Perm(gLock.moduleB_inst_Thread1.gLock, read)
52        ** Perm(gLock.moduleB_inst_Thread2.gLock, read)
53        ** gLock == gLock.channel_inst.gLock
54        ** gLock == gLock.moduleA_inst.gLock
55        ** gLock == gLock.moduleB_inst_State.gLock
56        ** gLock == gLock.moduleB_inst_Thread1.gLock
57        ** gLock == gLock.moduleB_inst_Thread2.gLock
58
59        // Read Permissions & Equality of Shared Module States
60        ** Perm(moduleB_inst_State, read)
61        ** Perm(gLock.moduleB_inst_State, read)
62        ** Perm(gLock.moduleB_inst_Thread1.moduleB_inst_State, read)
63        ** Perm(gLock.moduleB_inst_Thread2.moduleB_inst_State, read)
```

```
64        ** gLock.moduleB_inst_State == moduleB_inst_State
65        ** gLock.moduleB_inst_Thread2.moduleB_inst_State
66          == moduleB_inst_State
67
68        // Read Permissions & Equality of Shared Channels
69        ** Perm(gLock.channel_inst, read)
70        ** Perm(gLock.moduleA_inst.channelA, read)
71        ** Perm(moduleB_inst_State.channelB, read)
72        ** gLock.channel_inst == gLock.moduleA_inst.channelA
73        ** gLock.channel_inst == moduleB_inst_State.channelB
74
75
76        // Permissions, Null and Length Checks of further Heap Variables
77        ** Perm(moduleB_inst_State.numberOfPorts, write)
78        ** Perm(gLock.channel_inst.n, write)
79        ** Perm(gLock.channel_inst.buffer, read)
80        ** gLock.channel_inst.buffer != null
81        ** gLock.channel_inst.buffer.length == 42
82        ** Perm(gLock.channel_inst.buffer[*],write)
83        ;
84      while(true) {
85        moduleB_inst_State.returnZero(true);
86        moduleB_inst_State.channelB.writeToBuffer(42);   // Writes to Buffer
87      }
88
89      unlock gLock;
90    }
91  }
```

Listing A.9: PVL Module B Thread 1 (`ModuleBThread1.pvl`).

```
1   class ModuleBThread2 {
2
3     GlobalLock gLock;
4
5     ModuleBState moduleB_inst_State;
6
7     // NotNull Check of Arguments
8     requires globalLock != null;
9     requires moduleBState != null;
10    // Global Lock: Permissions and Value Checks
11    ensures Perm(gLock, read);
12    ensures gLock != null;
13    ensures gLock == globalLock;
14    // Shared ModuleBState: Permissions and Value Checks
15    ensures Perm(moduleB_inst_State, read);
16    ensures moduleB_inst_State != null;
17    ensures moduleB_inst_State == moduleBState;
18    ModuleBThread2(GlobalLock globalLock, ModuleBState moduleBState) {
19      gLock = globalLock;
```

```
20      moduleB_inst_State = moduleBState;
21    }
22
23
24    context Perm(gLock, read);
25    context gLock != null;
26    context Perm(gLock.moduleB_inst_Thread2, read);
27    context gLock.moduleB_inst_Thread2 == this;
28    void run () {
29      lock gLock;
30
31      loop_invariant true
32        // Permissions and Ownership of Global Lock
33        ** Perm(gLock, read)
34        ** gLock != null
35        ** held(gLock)
36
37         // Read Permissions for all Channel, State and Thread Classes
38        ** Perm(gLock.channel_inst, read)
39        ** Perm(gLock.moduleA_inst, read)
40        ** Perm(gLock.moduleB_inst_State, read)
41        ** Perm(gLock.moduleB_inst_Thread1, read)
42        ** Perm(gLock.moduleB_inst_Thread2, read)
43
44        // Equality of Global Lock's Thread1 and this
45        ** gLock.moduleB_inst_Thread2 == this
46
47        // Read Permissions & Equality of the Shared Global Lock
48        ** Perm(gLock.channel_inst.gLock, read)
49        ** Perm(gLock.moduleA_inst.gLock, read)
50        ** Perm(gLock.moduleB_inst_State.gLock, read)
51        ** Perm(gLock.moduleB_inst_Thread1.gLock, read)
52        ** Perm(gLock.moduleB_inst_Thread2.gLock, read)
53        ** gLock == gLock.channel_inst.gLock
54        ** gLock == gLock.moduleA_inst.gLock
55        ** gLock == gLock.moduleB_inst_State.gLock
56        ** gLock == gLock.moduleB_inst_Thread1.gLock
57        ** gLock == gLock.moduleB_inst_Thread2.gLock
58
59        // Read Permissions & Equality of Shared Module States
60        ** Perm(moduleB_inst_State, read)
61        ** Perm(gLock.moduleB_inst_State, read)
62        ** Perm(gLock.moduleB_inst_Thread1.moduleB_inst_State, read)
63        ** Perm(gLock.moduleB_inst_Thread2.moduleB_inst_State, read)
64        ** gLock.moduleB_inst_State == moduleB_inst_State
65        ** gLock.moduleB_inst_Thread1.moduleB_inst_State
66          == moduleB_inst_State
67
68        // Read Permissions & Equality of Shared Channels
```

```
      ** Perm(gLock.channel_inst, read)
      ** Perm(gLock.moduleA_inst.channelA, read)
      ** Perm(moduleB_inst_State.channelB, read)
      ** gLock.channel_inst == gLock.moduleA_inst.channelA
      ** gLock.channel_inst == moduleB_inst_State.channelB


      // Permissions, Null and Length Checks of further Heap Variables
      ** Perm(moduleB_inst_State.numberOfPorts, write)
      ** Perm(gLock.channel_inst.n, write)
      ** Perm(gLock.channel_inst.buffer, read)
      ** gLock.channel_inst.buffer != null
      ** gLock.channel_inst.buffer.length == 42
      ** Perm(gLock.channel_inst.buffer[*],write)
      ;
    while(true) {
      moduleB_inst_State.channelB.writeToBuffer(3);   // Writes to Buffer
    }

    unlock gLock;
  }
}
```

Listing A.10: PVL Module B Thread 2 (`ModuleBThread2.pvl`).

```
// No visilibity modifiers anywhere!
class Channel {

  GlobalLock gLock;

  // Define Directives
  int BUFFERSIZE;

  // Internal Data
  int n;
  int[] buffer;

  // Constructor
  // NotNull Check of Argument
  requires globalLock != null;
  // Global Lock: Permissions and Value Checks
  ensures Perm(gLock, read);
  ensures gLock != null;
  ensures gLock == globalLock;
  // Internal Data Fields: Permissions and Value Checks
  ensures Perm(n, write);
  ensures Perm(buffer, read);
  ensures buffer != null;
  ensures buffer.length == 42;
  ensures Perm(buffer[*],write);
```

```
26  Channel(GlobalLock globalLock) {
27    BUFFERSIZE = 42;
28
29    gLock = globalLock;
30    n = 0;
31    buffer = new int[BUFFERSIZE];
32   }
33
34
35  // Permissions and Ownership of Global Lock
36  context Perm(gLock, read);
37  context gLock != null;
38  context held(gLock);
39   // Read Permissions for all Channel, State and Thread Classes
40  context Perm(gLock.channel_inst, read);
41  context Perm(gLock.moduleA_inst, read);
42  context Perm(gLock.moduleB_inst_State, read);
43  context Perm(gLock.moduleB_inst_Thread1, read);
44  context Perm(gLock.moduleB_inst_Thread2, read);
45  // Equality of Global Lock's, moduleB's Channel and this
46  context Perm(gLock.channel_inst, read);
47  context Perm(gLock.moduleA_inst.channelA, read);
48  context Perm(gLock.moduleB_inst_State.channelB, read);
49  context gLock.channel_inst == this;
50  context gLock.moduleA_inst.channelA == this;
51  context gLock.moduleB_inst_State.channelB == this;
52  // Read Permissions & Equality of the Shared Global Lock
53  context Perm(gLock.channel_inst.gLock, read);
54  context Perm(gLock.moduleA_inst.gLock, read);
55  context Perm(gLock.moduleB_inst_State.gLock, read);
56  context Perm(gLock.moduleB_inst_Thread1.gLock, read);
57  context Perm(gLock.moduleB_inst_Thread2.gLock, read);
58  context gLock == gLock.channel_inst.gLock;
59  context gLock == gLock.moduleA_inst.gLock;
60  context gLock == gLock.moduleB_inst_State.gLock;
61  context gLock == gLock.moduleB_inst_Thread1.gLock;
62  context gLock == gLock.moduleB_inst_Thread2.gLock;
63  // Read Permissions & Equality of Shared Module States
64  context Perm(gLock.moduleB_inst_State, read);
65  context Perm(gLock.moduleB_inst_Thread1.moduleB_inst_State, read);
66  context Perm(gLock.moduleB_inst_Thread2.moduleB_inst_State, read);
67  context gLock.moduleB_inst_Thread1.moduleB_inst_State
68    == gLock.moduleB_inst_State;
69  context gLock.moduleB_inst_Thread2.moduleB_inst_State
70    == gLock.moduleB_inst_State;
71  // Permissions, Null and Length Checks of further Heap Variables
72  context Perm(gLock.moduleB_inst_State.numberOfPorts, write);
73  context Perm(gLock.channel_inst.n, write);
74  context Perm(gLock.channel_inst.buffer, read);
```

```
75  context gLock.channel_inst.buffer != null;
76  context gLock.channel_inst.buffer.length == 42;
77  context Perm(gLock.channel_inst.buffer[*],write);
78  void writeToBuffer(int c) {
79
80    loop_invariant true
81      ** Perm(gLock, read)
82      ** gLock != null
83      ** held(gLock)
84
85       // Read Permissions for all Channel, State and Thread Classes
86      ** Perm(gLock.channel_inst, read)
87      ** Perm(gLock.moduleA_inst, read)
88      ** Perm(gLock.moduleB_inst_State, read)
89      ** Perm(gLock.moduleB_inst_Thread1, read)
90      ** Perm(gLock.moduleB_inst_Thread2, read)
91
92      // Equality of Global Lock's, moduleB's Channel and this
93      ** Perm(gLock.channel_inst, read)
94      ** Perm(gLock.moduleA_inst.channelA, read)
95      ** Perm(gLock.moduleB_inst_State.channelB, read)
96      ** gLock.channel_inst == this
97      ** gLock.moduleA_inst.channelA == this
98      ** gLock.moduleB_inst_State.channelB == this
99
100     // Read Permissions & Equality of the Shared Global Lock
101     ** Perm(gLock.channel_inst.gLock, read)
102     ** Perm(gLock.moduleA_inst.gLock, read)
103     ** Perm(gLock.moduleB_inst_State.gLock, read)
104     ** Perm(gLock.moduleB_inst_Thread1.gLock, read)
105     ** Perm(gLock.moduleB_inst_Thread2.gLock, read)
106     ** gLock == gLock.channel_inst.gLock
107     ** gLock == gLock.moduleA_inst.gLock
108     ** gLock == gLock.moduleB_inst_State.gLock
109     ** gLock == gLock.moduleB_inst_Thread1.gLock
110     ** gLock == gLock.moduleB_inst_Thread2.gLock
111
112     // Read Permissions & Equality of Shared Module States
113     ** Perm(gLock.moduleB_inst_State, read)
114     ** Perm(gLock.moduleB_inst_Thread1.moduleB_inst_State, read)
115     ** Perm(gLock.moduleB_inst_Thread2.moduleB_inst_State, read)
116     ** gLock.moduleB_inst_Thread1.moduleB_inst_State
117       == gLock.moduleB_inst_State
118     ** gLock.moduleB_inst_Thread2.moduleB_inst_State
119       == gLock.moduleB_inst_State
120
121     // Permissions, Null and Length Checks of further Heap Variables
122     ** Perm(gLock.moduleB_inst_State.numberOfPorts, write)
123     ** Perm(gLock.channel_inst.n, write)
```

```
124       ** Perm(gLock.channel_inst.buffer, read)
125       ** gLock.channel_inst.buffer != null
126       ** gLock.channel_inst.buffer.length == 42
127       ** Perm(gLock.channel_inst.buffer[*],write)
128     ;
129     while (n == buffer.length) {
130        wait gLock;
131     }
132
133     buffer[17] = c;
134     n++;
135   }
136
137   // Permissions and Ownership of Global Lock
138   context Perm(gLock, read);
139   context gLock != null;
140   context held(gLock);
141    // Read Permissions for all Channel, State and Thread Classes
142   context Perm(gLock.channel_inst, read);
143   context Perm(gLock.moduleA_inst, read);
144   context Perm(gLock.moduleB_inst_State, read);
145   context Perm(gLock.moduleB_inst_Thread1, read);
146   context Perm(gLock.moduleB_inst_Thread2, read);
147   // Equality of Global Lock's, moduleB's Channel and this
148   context Perm(gLock.channel_inst, read);
149   context Perm(gLock.moduleA_inst.channelA, read);
150   context Perm(gLock.moduleB_inst_State.channelB, read);
151   context gLock.channel_inst == this;
152   context gLock.moduleA_inst.channelA == this;
153   context gLock.moduleB_inst_State.channelB == this;
154   // Read Permissions & Equality of the Shared Global Lock
155   context Perm(gLock.channel_inst.gLock, read);
156   context Perm(gLock.moduleA_inst.gLock, read);
157   context Perm(gLock.moduleB_inst_State.gLock, read);
158   context Perm(gLock.moduleB_inst_Thread1.gLock, read);
159   context Perm(gLock.moduleB_inst_Thread2.gLock, read);
160   context gLock == gLock.channel_inst.gLock;
161   context gLock == gLock.moduleA_inst.gLock;
162   context gLock == gLock.moduleB_inst_State.gLock;
163   context gLock == gLock.moduleB_inst_Thread1.gLock;
164   context gLock == gLock.moduleB_inst_Thread2.gLock;
165   // Read Permissions & Equality of Shared Module States
166   context Perm(gLock.moduleB_inst_State, read);
167   context Perm(gLock.moduleB_inst_Thread1.moduleB_inst_State, read);
168   context Perm(gLock.moduleB_inst_Thread2.moduleB_inst_State, read);
169   context gLock.moduleB_inst_Thread1.moduleB_inst_State
170     == gLock.moduleB_inst_State;
171   context gLock.moduleB_inst_Thread2.moduleB_inst_State
172     == gLock.moduleB_inst_State;
```

```
173   // Permissions, Null and Length Checks of further Heap Variables
174   context Perm(gLock.moduleB_inst_State.numberOfPorts, write);
175   context Perm(gLock.channel_inst.n, write);
176   context Perm(gLock.channel_inst.buffer, read);
177   context gLock.channel_inst.buffer != null;
178   context gLock.channel_inst.buffer.length == 42;
179   context Perm(gLock.channel_inst.buffer[*],write);
180   int returnElemNum() {
181     n--;
182     notify gLock;
183
184     return n;
185   }
186 }
```

Listing A.11: PVL Channel (`Channel.pvl`).

List of Abbreviations

AMBA Advanced Microcontroller Bus Architecture 14

AMS Analog/Mixed-Signal 12

ASIC Application-Specific Integrated Circuit 10

AST Abstract Syntax Tree 27

COL Common Object Language 27

CSL Concurrent Separation Logic 2

DSP Programmable Digital Signal Processor 10

FIFO First-In–First-Out Queue 14

HDL Hardware Description Language 11

IDF Implicit Dynamic Frames 28

ISA Instruction Set Architecture 10

JML Java Modeling Language 32

LRM Language Reference Manual 23

PVL Prototypal Verification Language 2

RTL Register-Transfer Level 10

SAT Boolean Satisfiability 5

SMT Satisfiability Modulo Theories 5

List of Figures

List of Tables

Listings

Bibliography

[ABH16] Afshin Amighi, Stefan Blom, and Marieke Huisman. "VerCors: A layered approach to practical verification of concurrent software". In: *2016 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*. IEEE. 2016, pp. 495–503 (cit. on p. 28).

[Acc21a] Accellera Systems Initiative. *SystemC Standards*. URL: `https://www.accellera.org/downloads/standards/systemc` (visited on 08/07/2021) (cit. on p. 12).

[Acc21c] Accellera Systems Initiative. *About SystemC*. URL: `https://www.accellera.org/community/systemc/about-systemc` (visited on 08/10/2021) (cit. on p. 12).

[Ami+12] Afshin Amighi, Stefan Blom, Marieke Huisman, and Marina Zaharieva-Stojanovski. "The VerCors project: Setting up basecamp". In: *Proceedings of the sixth workshop on Programming languages meets program verification*. ACM, 2012, pp. 71–82 (cit. on p. 28).

[BD04] David C. Black and Jack Donovan. *SystemC: From the Ground Up*. Springer, 2004 (cit. on pp. 11, 14).

[BDH15] Stefan Blom, Saeed Darabi, and Marieke Huisman. "Verification of loop parallelisations". In: *International conference on fundamental approaches to software engineering*. Springer. 2015, pp. 202–217 (cit. on p. 40).

[BK07] Bernhard Beckert and Vladimir Klebanov. "A dynamic logic for deductive verification of concurrent Java programs with condition variables". In: *Satellite Workshop at CONCUR 2007*. 2007, p. 3 (cit. on p. 65).

[BO16] Stephen Brookes and Peter W. O'Hearn. "Concurrent Separation Logic". In: *ACM SIGLOG News* 3.3 (2016), pp. 47–65 (cit. on pp. 2, 28, 43).

[BST+10] Clark Barrett, Aaron Stump, Cesare Tinelli, et al. "The smt-lib standard: Version 2.0". In: *Proceedings of the 8th international workshop on satisfiability modulo theories (Edinburgh, England)*. Vol. 13. 2010, p. 14 (cit. on p. 28).

[Cho+10] Chun-Nan Chou, Chang-Hong Hsu, Yueh-Tung Chao, and Chung-Yang Huang. "Formal deadlock checking on high-level SystemC designs". In: *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE. 2010, pp. 794–799 (cit. on p. 6).

[DB08] Leonardo De Moura and Nikolaj Björner. "Z3: An efficient SMT solver". In: *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2008, pp. 337–340 (cit. on pp. 28, 43).

[DG97] Giovanni De Micheli and Rajesh K. Gupta. "Hardware/Software Co-Design". In: *Proceedings of the IEEE* 85.3 (1997), pp. 349–365 (cit. on pp. 2, 9, 10).

[Dij72] Edsger W. Dijkstra. "The Humble Programmer". In: *Communications of the ACM* 15.10 (1972), pp. 859–866 (cit. on p. 1).

[Grö+02] T. Grötker, S. Liao, G. Martin, and S. Swan. *System Design with SystemC™*. Springer US, 2002 (cit. on p. 11).

[Haa+14] Christian Haack, Marieke Huisman, Clément Hurlin, and Afshin Amighi. "Permission-based separation logic for multithreaded Java programs". In: *arXiv preprint arXiv:1411.0851* (2014) (cit. on p. 40).

[Her+18] Vladimir Herdt, Hoang M. Le, Daniel Grosse, and Rolf Drechsler. "Verifying SystemC using intermediate verification language and stateful symbolic simulation". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 38.7 (2018), pp. 1359–1372 (cit. on p. 7).

[Her10] Paula Herber. *A Framework for Automated HW/SW Co-Verification of SystemC Designs using Timed Automata*. Logos Verlag Berlin GmbH, 2010 (cit. on p. 25).

[Her14] Paula Herber. "The RESCUE Approach - Towards Compositional Hardware/-Software Co-verification". In: IEEE. 2014, pp. 721–724 (cit. on p. 6).

[HFG08] Paula Herber, Joachim Fellmuth, and Sabine Glesner. "Model checking SystemC designs using timed automata". In: IEEE. 2008, pp. 131–136 (cit. on pp. 5, 71).

[HGD21] Vladimir Herdt, Daniel Grosse, and Rolf Drechsler. "Formal Verification of SystemC-Based Designs using Symbolic Simulation". In: *Enhanced Virtual Prototyping*. Springer, 2021, pp. 59–117 (cit. on p. 7).

[HH14] Paula Herber and Bettina Hünnemeyer. "Formal Verification of SystemC Designs using the BLAST Software Model Checker". In: *ACESMB@ MoDELS*. 2014, pp. 44–53 (cit. on p. 6).

[HL20] Paula Herber and Timm Liebrenz. "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)*. IEEE. 2020, pp. 1–6 (cit. on p. 7).

[HPG15] Paula Herber, Marcel Pockrandt, and Sabine Glesner. "STATE – A SystemC to Timed Automata Transformation Engine". In: IEEE. 2015, pp. 1074–1077 (cit. on p. 6).

[IEEE11] Accellera Systems Initiative et al. "IEEE Standard for Standard SystemC® Language Reference Manual". In: *IEEE Std 1666–2011 (Revision of IEEE Std 1666–2005)* (2011) (cit. on pp. 12, 17, 23, 47).

[JH15] Lydia Jass and Paula Herber. "Bit-Precise Formal Verification for SystemC Using Satisfiability Modulo Theories Solving". In: *International Embedded Systems Symposium*. Springer. 2015, pp. 51–63 (cit. on p. 7).

[KEP06] Daniel Karlsson, Petru Eles, and Zebo Peng. "Formal verification of SystemC designs using a Petri-net based representation". In: *Proceedings of the Design Automation & Test in Europe Conference*. Vol. 1. IEEE. 2006, pp. 1–6 (cit. on p. 6).

[KMS12] Ioannis T Kassios, Peter Müller, and Malte Schwerhoff. "Comparing verification condition generation with symbolic execution: an experience report". In: *International Conference on Verified Software: Tools, Theories, Experiments*. Springer. 2012, pp. 196–208 (cit. on p. 28).

[KS05] Daniel Kroening and Natasha Sharygina. "Formal verification of SystemC by automatic hardware/software partitioning". In: *Proceedings. Second ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2005. MEMOCODE'05.* IEEE. 2005, pp. 101–110 (cit. on p. 6).

[LBR98] Gary T Leavens, Albert L Baker, and Clyde Ruby. "JML: a Java modeling language". In: *Formal Underpinnings of Java Workshop (at OOPSLA'98)*. Citeseer. 1998, pp. 404–420 (cit. on p. 32).

[Le+13] Hoang M Le, Daniel Grosse, Vladimir Herdt, and Rolf Drechsler. "Verifying SystemC using an intermediate verification language and symbolic simulation". In: *Proceedings of the 50th Annual Design Automation Conference*. 2013, pp. 1–6 (cit. on p. 7).

[Lee02] Edward A. Lee. "Embedded Software". In: *Advances in Computers*. Vol. 56. Elsevier, 2002, pp. 55–95 (cit. on pp. 2, 12).

[LX20] Bin Lin and Fei Xie. "A Systematic Investigation of State-of-the-Art SystemC Verification". In: *Journal of Circuits, Systems and Computers* 29.15 (2020) (cit. on p. 8).

[MK06] Jeff Magee and Jeff Kramer. *Concurrency: State Models and Java Programs*. 2nd. Wiley Publishing, 2006 (cit. on p. 67).

[MSS16] P. Müller, M. Schwerhoff, and A. J. Summers. "Viper: A Verification Infrastructure for Permission-Based Reasoning". In: *Verification, Model Checking, and Abstract Interpretation (VMCAI)*. Ed. by B. Jobstmann and K. R. M. Leino. Vol. 9583. LNCS. Springer-Verlag, 2016, pp. 41–62 (cit. on p. 26).

[ORe17] Gerard O'Regan. *Concise Guide to Formal Methods*. Springer, 2017 (cit. on p. 1).

[RS20] Jürgen Reichardt and Bernd Schwarz. *VHDL-Simulation und -Synthese*. De Gruyter Oldenbourg, 2020 (cit. on p. 11).

[Rub20] RB Rubbens. "Improving Support for Java Exceptions and Inheritance in VerCors". MA thesis. University of Twente, 2020 (cit. on p. 47).

[SAVES21] Formal Methods and Tools Group (UT) and Embedded Systems Group (WWU). *SAVES: Scalable Verification of Industriaal Embedded Control Systems*. url: https://www.utwente.nl/en/eemcs/fmt/research/projects/saves/ (visited on 09/27/2021) (cit. on p. 66).

[Sch16] Malte H Schwerhoff. "Advancing Automated, Permission-Based Program Verification Using Symbolic Execution". PhD thesis. ETH Zurich, 2016 (cit. on p. 28).

[SH20] Simon Schwan and Paula Herber. "Optimized Hardware/Software Co-Verification using the UCLID Satisfiability Modulo Theory Solver". In: *IEEE 29th International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE)*. IEEE. 2020, pp. 225–230 (cit. on pp. 7, 8).

[SJP09] Jan Smans, Bart Jacobs, and Frank Piessens. "Implicit dynamic frames: Combining dynamic frames and separation logic". In: *European Conference on Object-Oriented Programming*. Springer. 2009, pp. 148–172 (cit. on pp. 28, 43).

[TM08] Donald Thomas and Philip Moorby. *The Verilog® hardware description language*. Springer Science & Business Media, 2008 (cit. on p. 11).

[Ver21] Formal Methods and Tools Group, University of Twente. *The VerCors Verifier Online*. url: https://vercors.ewi.utwente.nl/ (visited on 08/07/2021) (cit. on pp. 2, 26, 27, 31, 34, 109).

[Ver21g] Formal Methods and Tools Group, University of Twente. *VerCors Source Code on GitHub*. url: https://github.com/utwente-fmt/vercors (visited on 08/29/2021) (cit. on p. 27).

[Vip21] Programming Methodology Group, ETH Zürich. *Viper Online*. url: http://viper.ethz.ch (visited on 08/31/2021) (cit. on pp. 28, 43).

Declaration of Academic Integrity

I, *Stefanie Eva Drerup*, hereby confirm that this thesis on

> *Deductive Verification of Integrated Hardware/Software Systems with the VerCors Verification Tool*

is solely my own work and that I have used no sources or aids other than the ones stated. All passages in my thesis for which other sources, including electronic media, have been used, be it direct quotes or content references, have been acknowledged as such and the sources cited.

Münster, September 30, 2021

Stefanie Eva Drerup

I agree with a comparison of the thesis with other texts in order to find matches and with the storage of this thesis in a database for this purpose.

Münster, September 30, 2021

Stefanie Eva Drerup

Eidesstattliche Erklärung

Hiermit versichere ich, *Stefanie Eva Drerup*, dass die vorliegende Arbeit über

> *Deductive Verification of Integrated Hardware/Software Systems*
> *with the VerCors Verification Tool*

selbstständig verfasst worden ist, dass keine anderen Quellen und Hilfsmittel als die angegebenen benutzt worden sind und dass die Stellen der Arbeit, die anderen Werken – auch elektronischen Medien – dem Wortlaut oder Sinn nach entnommen wurden, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht worden sind.

Münster, 30. September 2021

Stefanie Eva Drerup

Ich erkläre mich mit einem Abgleich der Arbeit mit anderen Texten zwecks Auffindung von Übereinstimmungen sowie mit einer zu diesem Zweck vorzunehmenden Speicherung der Arbeit in einer Datenbank einverstanden.

Münster, 30. September 2021

Stefanie Eva Drerup