FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT

Lehrstuhl für Informatik 3

Rechnerarchitektur

Andreas Schärtl

Design and Implementation of an Editor for Creating and Simulating Heterogeneous Systems

Bachelorarbeit im Fach Informatik

16. Januar 2019

Please cite as:
Andreas Schärtl, “Design and Implementation of an Editor for Creating and Simulating Heterogeneous Systems,”
Bachelor’s Thesis (Bachelorarbeit), University of Erlangen, Dept. of Computer Science, January 2019.

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG

Friedrich-Alexander-Universität Erlangen-Nürnberg
Department Informatik
Rechnerarchitektur

Martensstr. 3 · 91058 Erlangen · Germany

www3.cs.fau.de

Design and Implementation of an Editor for Creating and Simulating Heterogeneous Systems

Bachelorarbeit im Fach Informatik

vorgelegt von

Andreas Schärtl

geb. am 01. Mai 1992
in Nabburg

angefertigt am

Lehrstuhl für Informatik 3
Rechnerarchitektur

Department Informatik
Friedrich-Alexander-Universität Erlangen-Nürnberg

Betreuer:	**Sebastian Rachuj**
	Dr.-Ing. Marc Reichenbach
Betreuender Hochschullehrer:	**Prof. Dr.-Ing. Dietmar Fey**

Beginn der Arbeit:	**19. Juli 2018**
Abgabe der Arbeit:	**16. Januar 2019**

Erklärung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde.
Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Declaration

I declare that the work is entirely my own and was produced with no assistance from third parties.
I certify that the work has not been submitted in the same or any similar form for assessment to any other examining body and all references, direct and indirect, are indicated as such and have been cited accordingly.

(Andreas Schärtl)
Erlangen, 16. Januar 2019

Zusammenfassung

Beim Design von heterogenen Systems-On-A-Chip (SoC) kommen unterschiedliche CPU- und GPU-Simulatoren zum Einsatz. Anwendungen finden sich in den verschiedensten Gebieten, unter Anderem bei mobilen Endgeräten, Fahrzeugen oder IoT-Anwendungen. Zur Entwicklung und Evaluation von SoCs soll in dieser Arbeit ein Framework, *Kras*, entwickelt werden, welches existierende Simulatoren vereinigt.

Kras basiert auf SystemC, einer Erweiterung der Programmiersprache C++, entwickelt für funktionale Simulation von Hardwarekomponenten. Simulationen werden durch INI-Dateien beschreiben und zu SystemC übersetzt. Um dann die Simulation laufen zu lassen, wir dieser SystemC-Code ausgeführt. Diese Arbeit zeigt, dass es möglich ist, verschiedene Simulationssoftware mit SystemC zu kombinieren. Der Preis hierfür sind Simulationen, die bis zu drei Größenordnungen langsamer in der Ausführung sein können.

Abstract

During work on heterogeneous systems on a chip (SoC) designs, different CPU and GPU simulators are used. Applications for such systems are found in many areas, including mobile phones, the automotive sector and IoT solutions. This thesis wants to develop a framework for design and evaluation of SoCs, *Kras*, that combines different existing simulators into one package.

Kras is built around the SystemC environment, an extension to the C++ programming language for functional simulation of hardware components. Simulations are configured in INI files which are compiled to SystemC code. Running this SystemC code means running the simulation. This thesis shows that it is possible to combine existing simulator software using SystemC and a unified description language. The disadvantage of this approach are slowdowns during simulation that can be as big as three orders of magnitude.

Contents

Chapter 1

Introduction

Today, systems on a chip are common in many applications and focus of much research. Simulations are used in many stages of development of these systems. As such, it comes at no surprise that a lot of simulation software is now available, all with individual properties and strengths. On the downside, it often can be hard to combine these simulators, requiring duplicate work to use the distinct advantages of different simulation packages.

This thesis tries to find a way to design a flexible framework for configuring simulations that work across different simulators. The framework should include support for a graphical editor that allows users to edit simulation setups in an interactive fashion.

1.1 Context

Systems on a Chip (SoC) are a class of integrated circuits that combine one or more general purpose processing (CPU) cores with other components on one die. Previously, these components would have been implemented as distinct integrated circuits on a circuit board. Today, they are all integrated in one chip. Such components include bus systems, memory, graphic processors (GPUs) and digital signal processors [1]. Integrating all these components into one chip has advantages: For one, (1) SoCs save space compared to multiple distinct circuits. This is necessary for mobile and internet of things (IoT) applications where space can be constrained. SoCs can also be (2) cheaper to produce compared to using distinct chips: Instead of fabricating different distinct chips and then wiring these chips together, a SoC can be manufactured as a single unit. [1, 2].

SoCs make it easy to combine components (e.g. CPU, memory) on one die. This is because different components do not get implemented from scratch, rather designer use pre-made components (*intellectual property blocks*, *IP blocks*) provided

by various hardware and software vendors. Combining these IP blocks results in a digital design that can then be manufactured by a semiconductor lab. This relative ease of development means that SoCs can be fine-tuned for many applications [3, 4].

A next step towards more flexible systems are so-called *heterogeneous systems*, which extend the idea of the SoC. A heterogeneous system can include multiple CPUs with different properties or architectures dedicated to specific tasks. For example, a notebook computer might contain two kinds of CPUs: A high-performance, high-consumption, CPU on one hand and a low-power, low-consumption, CPU on the other hand. While plugged into wall power, the high-performance CPU provides quick computation, while on battery the low-power CPU improves battery life. Both CPUs reside on the same chip, manufactured as a single unit, keeping production cost low [5]. This illustrates how heterogeneous systems can offer more flexibility compared to traditional systems.

The only way to evaluate SoC designs in a reasonable amount of time and at a justifiable cost is to use simulations. Simulations are based around models of the components that make the design as well as the interconnects between them. Simulations provide detailed information about functionality and timing properties. This information is important to avoid design mistakes early in development [6]. For this purpose, a broad range of simulation tools has emerged.

1.2 Problem

Different simulation software exhibits different properties. Differences are, among others, in feature set, accuracy and simulation speed. All these different properties are worth considering during design. Ideally, an SoC designer uses the most accurate simulator to gather timing information and the quickest simulator to do functional testing.

Unfortunately, using different simulators for the same project can be cumbersome and prone to error. For one, (1) configurations are incompatible with each other. A configuration built for one simulator will not run on another simulator without change. In consequence, the layout of the SoC design needs to be re-created many times, just so the semantically identical model is understood by a different simulator. During development, inevitable changes to the layout need to be replicated as well, once for every simulation package. Another problem is that (2) models of components might be incompatible between simulators. A component such as a bus connect or RAM will have to be re-created once for every simulation package. Both (1) and (2) stand in stark contrast to the "*don't repeat yourself*" (DRY) principle [7, p.

27] which adds uncertainty to the design process: Provided multiple configurations for the same SoC, do these configurations model the same chip?

This thesis wants to develop and evaluate an approach for unified system simulation. For this, this thesis introduces the *Kras* framework. Kras can configure different CPU and GPU simulators as well as peripheral components based on a unified configuration file. The framework comes together with a graphical front-end, such that even big designs with multiple connections and components can be configured without losing track of the general setup.

1.3 Current Approaches

The problem of a diverse set of simulation software is not a new one. Existing modeling software packages try to ensure some degree of cooperability with other tools and maintain a reasonable level of complexity such that the user does not get overwhelmed.

1.3.1 Virtual Platforms

One category of simulation software is the "*virtual platform*" [8]. A virtual platform provides the user with a set of hardware models (e.g. processors, memory, bus systems and IO peripherals) combined in one integrated editor. These components can typically be configured using a graphical design tool [8]. Ideally, users can complete the whole design within a given virtual platform, with all necessary tools in one place.

One example of such a software is *Platform Architect* [9], a commercial product distributed by Synopsis. The main interface consist of a graphical editor that lets users place components into the simulation, configure interconnects between these components, including memory and bus configurations. Platform Architect is based around SystemC, a publicly available extension of the C++ programming language [10]. This means that other components implemented in SystemC, including full CPU cores, can be integrated into a Platform Architect workflow.

Vista from Mentor Graphics is a tool similar to Platform Architect [11, p. 5]. It provides capabilities for testing abstract models of SoC designs. Just like Platform Architect is based on the SystemC language; included models of processors and peripherals can be extended with third party components. Configuration of a simulation is handled in a schematic view for which corresponding SystemC code is compiled and executed for simulation. These simulation binaries can be executed either (1) directly through the graphical interface or (2) automated in scripts [12].

Virtual platforms can be a convenient environment for SoC simulation. Being built around an established language, such as SystemC, means that Platform Architect and

Vista can be used in conjunction with third party models. A possible disadvantage of virtual platforms is that running everything through one big tool creates a dependency on a tool vendor: While it might be possible to integrate custom designs for the virtual platform, it might not be possible to integrate the vendors' models into other tools. One big all-combining tool might also not cooperate as well in automated setups and continuous integration. It is true that integrated virtual platforms can be very productive and appropriate for many designs. However, lack of flexibility and dependency on a single product, potentially very expensive, can be a problem.

1.3.2 Standardization Between Tools

Effort has been made to design a format for describing the properties of hardware models in a universal manner. The outcome of this effort is *IP-XACT*, an IEEE standard [13]. It aims to provide a standardized way of describing hardware models using XML. The standard contains schema for defining the interface of hardware components (e.g. ports, available registers), as well as the interconnects and bus configurations of a single component or system. These standardized components can then be integrated into full setups. The idea is that IP-XACT descriptions represent a platform-agnostic representation of a simulation setup. However, IP-XACT is mostly concerned with metadata, that is the interface of models, but not their functional behavior [8, 14]. Functional behavior has to be implemented by the tool running the simulation. Porting a model to another simulator would still mean re-implementing the models behavior.

A language that describes interface as well as functionality of hardware components is SystemC [10]. SystemC is not a stand-alone language, rather it is a library for the C++ programming language designed for functional modeling of different systems. SystemC code is compiled using a regular C++ compiler and linked against the SystemC library. This results in a binary that runs a simulation of the configured system. This is illustrated in Figure 1.1. As desired by the developer, this simulation

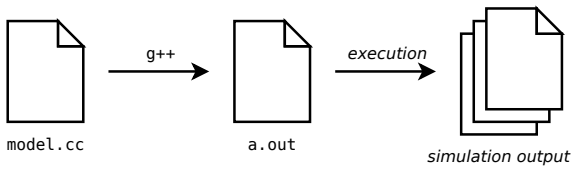


Figure 1.1 – How SystemC runs simulations. Individual models are combined in a top level file `model.cc` which is compiled using a regular C++ compiler. The resulting executable is run as a normal program and outputs logging information as configured.

binary prints functional or timing analysis to the console or writes it to log files. The SystemC standard comes together with an Apache-licensed reference implementation [15] and has been adopted by various design tools (Synopsis Platform Architect, Mentor Graphics Vista). This means that models encoded in SystemC will be able to run on various simulation software targets.

1.3.3 Schematic Views

One similarly between many chip design tools is the use of a schematic user interface that lets users change the configuration of a simulation by dropping components into a canvas and then connecting these components which each other. For example, both Synopsis Platform Architect and Mentor Graphics Vista offer such interfaces, as well as MathWorks' *HDL Verifier* [16] hardware test benching tool or the ARM *SoC Designer* [17], a tool for prototyping SoCs based around ARM processors.

1.4 Goals

Motivated by the problem statement and based on previous work, three goals were made out for the simulation framework designed in this thesis:

- There are many simulation packages available to a designer. To avoid code duplication and conform to the DRY-principle, it needs to be possible to integrate Kras with many different simulation tool chains. The framework needs to be *extensible*. It should be able to support various models of CPUs and peripherals.
- While different simulators from different vendors do have different properties, their core interface should be the same. The framework should *work with abstract descriptions of simulation layouts*. Implementation details should be transparent to the user when possible, such that the SoC architecture can be the focus.
- Kras combines different tools that speak different languages and controls all these tools using a single interface. This is necessary because simulation software vendors designed products not necessarily compatible to each other. As a layer built on top of this software, Kras should not commit the same mistake. Instead, it should be a loosely coupled [18, p. 30] component of a simulation tool chain that cooperates with other tools and, if necessary, could be easily replaced. Kras should be easy to incorporate the tool into other workflows. The framework should offer *cooperability*.

Using different simulation software offers advantages, but can be cumbersome. The goal of this thesis is to develop and evaluate Kras, a framework which (1) simu-

lators can easily be adapted to (extensibility), (2) allows the designer to focus on the design and not simulation details (is abstract) and (3) is not difficult to use with other tools and individual workflows (cooperability).

Chapter 2

Fundamentals

This chapter presents the fundamental building blocks that make up the base of Kras. First, section 2.1 introduces the different abstraction levels of SoC simulation The abstract transaction level modeling (TLM) proves to be the right level of simulation for this project. A popular implementation of TLM is found in the SystemC language, introduced in section 2.2. Finally, section 2.3 presents a selection of simulation packages. Combining these simulators using the SystemC language will be the main task of Kras. These are the fundamentals this thesis is built upon.

2.1 Simulation in Computer Architecture Design

Before starting the actual discussion, it is helpful to make some definitions. Based on previous work found in literature [19, p. 6ff], these definitions help to make this thesis more clear. A *model* is a representation of a real object (e.g. a CPU core, a bus system, an IO device). The model encapsulates the distinct properties of the original object (e.g. functional behavior, timing information) in the form of a mathematical representation or program code. A *configuration* shall describe the (1) setup of models representing real components and (2) the interconnects between those components. Finally, a *simulation* executes a configuration. The simulation takes the modeled components together with their interconnects, supplies the components with input and records the responses of the components to the simulated stimuli. To

Simulations are an integral part to SoC design and evaluation. For the system designer, multiple categories of simulations are available. These categories differ in level of abstraction and available primitives. This thesis differentiates between two categories, *register transfer level* (RTL) models and *transaction level modeling* (TLM) [20]. Development typically starts at a very abstract level (TLM) and eventually progresses to a point where individual logic gates are placed (RTL) [21].

The remainder of this section will introduce both RTL and TLM in more detail and explain why TLM is the right level of abstraction for Kras.

2.1.1 Register Transfer Level

Models at the register transfer level (RTL) are based on two main building blocks, (1) memory (registers) and (2) logic (transfer between registers) [22] as exemplified in figure 2.1a. These primitive building blocks result in a very low level of abstraction close to real silicon circuits.

Development of custom silicon requires RTL modeling during design [21]. However, in the design of SoCs or heterogeneous systems, RTL modeling generally only plays a role during the end of development. Before RTL simulation, a lot of time is spent with more abstract design exploration. The core building blocks here are CPU cores, signal processors and bus systems, not registers and logic gates. While it is possible to represent these abstract components as RTL models, this is not a good solution.

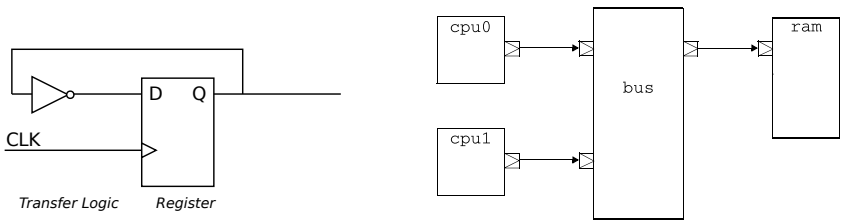
For one, RTL models are (1) too detailed during design exploration. They do not offer the level of productivity required in today's competitive market and delay projects with unnecessary detail [23]. RTL models are too complex, they don't allow quick iteration because designers get overwhelmed by the fine detail. It is not only the designer that is overwhelmed by this level of complexity, (2) the high level of detail also results in very demanding computer simulations that take up a lot of time to run. While it is possible to boot a Linux kernel in an RTL simulation of a processor, this simulation might take more than ten hours [24]. In the end this again means unacceptable waiting periods in SoC design.

The high level of detail found in RTL models results in unacceptable mental load and long-winded computer simulation. They are not a viable choice for abstract SoC design. As such, RTL level modeling is too detailed for the intents of Kras, a framework that wants to take a more abstract perspective.

2.1.2 Transaction Level Modeling

RTL models are too fine grained because of insurmountable detail. On the other hand, transaction level modeling (TLM) offers a more abstract point of view on a system. Instead of thinking of systems in terms of register memory and logic gates, TLM models are made up of abstract components that represent real functionality on a chip. 2.1b illustrates this with a basic TLM schematic. Set up in a configuration, this creates an abstract model of a system that can be simulated to gain approximations of performance and functionality [20].

TLM simulations do not simulate voltage levels on individual wires, rather they express the idea of communication in a complex system. During simulation, the



(a) RTL schematic **(b)** TLM schematic

Figure 2.1 – Differences in schematics depending on abstraction level. RTL schematics consists of registers and transfer logic while the building blocks in TLM schematics are abstract components. A line in the RTL schematic represents a wire on the circuit while a line in a TLM schematic represents a path for messages.

individual components of a TLM model exchange messages, which is sufficient for functional simulation [23]. In an RTL model, individual lanes of the bus would have been modeled, not in the case of TLM.

This abstract way of thinking about systems has multiple consequences for TLM simulations. Naturally, the high level of abstraction results in inaccuracies regarding timing and potentially functional properties. However, for design exploration and development, such approximations often are sufficient [23]. On the other hand, TLM simulations come with advantages that make them attractive for use in SoC design, effectively negating the disadvantages of RTL. For one, (1) the high level of abstraction results in easier to understand configurations. A more simple configuration results in simulations less prone to error and higher productivity. The reduced complexity makes it easier to cope with the complexity of a modern SoC [25, p. 472ff]. The level of complexity has an effect on execution time as well. Because they do not have to model individual wires and voltage levels, (2) TLM simulations can achieve higher throughput compared to RTL simulations. This allows for quick iteration and change [24].

TLM offers just the right level of abstraction for Kras. For this reason, the work in this thesis only concerns itself with TLM simulation.

2.2 SystemC

TLM models can be implemented in general purpose programming languages, such as Java [26], but there already exist programming environments designed for use with TLM. A language that has good support for modeling TLM simulations is the

freely available SystemC language, standardized by the IEEE [10]. While SystemC does support simulation of RTL models, it also supports transaction level modeling in the form of the SystemC TLM-2.0 standard. This TLM extension is integrated into SystemC and now part of the regular SystemC release [10, 15].

2.2.1 Primitives of SystemC TLM

A TLM configuration consists of instances of models, each with a well-defined interface, and the interconnects between those instances. This subsection investigates how these primitives are implemented in SystemC TLM.

Models of hardware components are represented by SystemC *modules* [10]. For example, the code in Listing 2.1 declares a SystemC module named `Printer`. SystemC modules are regular C++ classes. As such, modules can contain member variables and methods. In this example, the module has one member variable `in` and a method `callback`. In the implementation of the constructor, not listed, the socket `in` is configured to call the method `callback` on events.

SystemC TLM modules define the interface of a component in the form of *sockets* which can be understood as interconnection points (ports) of a component. SystemC TLM differentiates between two kinds of sockets: On one hand, *initiator sockets* are sockets that can start communication while on the other hand, *target sockets* may only reply to messages sent from initiator sockets. Master components use initiator sockets to send requests to target sockets of slaves [10, p. 413ff].

To place a component into a configuration, an object of the corresponding module (class) is created as usual in C++. As such, creating an object means invoking the constructor of that class. This might require some constructor arguments. In particular, SystemC modules will usually have at least one mandatory constructor parameter (the name of the component; used for logging) [10, p. 57ff], but in general SystemC modules can have any number of constructor parameters. Additionally to constructor parameters, template parameters might be required to instantiate an object. Indeed this is often the case for modules part of the SystemC TLM library.

Modules encapsulate the individual properties of a component and define their interface using sockets. What is left is the interconnect between instances. In a

```
1 SC_MODULE (Printer) {
2     simple_target_socket<Printer, 32> in;
3
4     Printer(const sc_module_name &name);
5     void callback(tlm_generic_payload &p, sc_time &delay);
6 };
```

Listing 2.1 – SystemC code that declares a module `Printer`.

top level SystemC file that defines the configuration, modules are instantiated and connected using the `bind` method provided by the socket classes [10, p. 422ff]. Given an instantiated module `master` with initiator socket `initiator` and another instance `slave` with target socket `target`, binding them is done by calling `master.initiator.bind(slave.target)`. Alternatively, instead of binding the initiator to the target, the target can be bound to the initiator. Both cases are equivalent when dealing with TLM sockets, but regardless of who binds who, only initiators can begin communication.

2.2.2 Simulation

Now that all primitives are in place, a simulation can be configured. In a top level file, a function `sc_main`, analogous to the regular `main` function, is defined. This will be the entry point of the simulation. In `sc_main`, the individual modules are instantiated as objects and bound together using `bind` calls. This concludes the initial configuration and is referred to as *elaboration* [10, p. 12ff]. After elaboration, logging and diagnostics can be configured. Finally, the simulation is started using the `sc_start` function [10, p. 21]. Depending on the arguments passed to `sc_start`, the simulation runs until a certain amount of time has been simulated or until the program is interrupted by the user.

During simulation, master modules will initiate communication with slave modules using the master's initiator socket. Passing a message M is implemented as a function call on the initiator socket. In turn, the initiator socket forwards M to the target socket it is bound to by calling a method on that target socket. This method is the callback which simulates the behavior of the slave component to message M. A reply is generated and then forwarded to the original sender in similar fashion.

In summary, modules are used to model components as C++ classes. Their interface is defined using sockets which are connected using `bind` calls. During simulation, messages are passed, simulating the behavior of the configured system.

2.2.3 Use in this Thesis

With a better understanding of SystemC, now for the reasons why SystemC was chosen for use with Kras.

- One of the goals of the Kras framework is extensibility. SystemC is a publicly available standard [10] with an open source reference implementation available [15].
- TLM is the right level of abstraction for Kras and SystemC has native support for it. Using an existing framework for TLM means that primitives and concepts

do not need to be re-implemented from scratch. The entire simulation logic (`sc_main`, logging) is already implemented by SystemC.

- SystemC is a C++ library, simulations are just regular executables. They are easy to incorporate in other workflows.

2.3 Available Components

SystemC acts as glue between different simulation packages. These packages provide models of CPU cores or entire GPUs. As previously mentioned, there are many such models available. This section takes a closer look at two packages that provide CPU models, *gem5* and *OVPsim*. Then, the GPU simulator *GPGPU-Sim* with support for CUDA API is introduced. For each individual component, this section discusses core features, interface and integration possibilities with SystemC.

2.3.1 gem5

The freely available gem5 simulator is described as a "simulation framework" [27]. At its core are CPU models with support for a a rich set of instruction set architectures (ISAs), including ARM, MIPS, Power, and x86 with the new RISC-V architecture being worked on [28]. The different gem5 CPU models are characterized by a trade-off between accuracy and simulation speed: On one end of the spectrum is the *AtomicSimple* model which simplifies simulation by modeling a single instruction per cycle which is sufficient for functional evaluation. At the other extreme of the spectrum is the more detailed and as such more demanding *O3* model. It represents a pipelined out-of-order CPU more in line with modern real-life CPUs [27].

As gem5 is a framework, there are various other features as well, including simulation capabilities for sophisticated memory configurations and whole systems. These additional features are flexible and can be enabled and disabled as desired [27]. Overall, gem5 offers a number of CPU models at various levels of abstractions. Additional features are available but optional.

Usually, gem5 is configured using the Python programming language. All simulation objects of gem5 are exposed both as C++ and Python classes that inherit from a `SimObject` base class. To set up a simulation, a Python program initializes the various components as Python objects. These objects are then connected by setting object variables of the components to reference other components. A small extract of a possible configuration file is provided in Listing 2.2.

gem5 does not use SystemC to represent models, but a compatibility layer between gem5 and SystemC TLM was added in 2017 [30]. This compatibility layer lets

```
# Create a simple CPU
system.cpu = TimingSimpleCPU()

# Create a memory bus, a system crossbar, in this case
system.membus = SystemXBar()

# Hook the CPU ports up to the membus
system.cpu.icache_port = system.membus.slave
system.cpu.dcache_port = system.membus.slave
```

Listing 2.2 – Part of a gem5 configuration written in Python. First, a CPU core is created. This core is then connected to a bus system. Code taken from the gem5 project [29].

users combine gem5 components with SystemC modules. Integration with SystemC, and therefore Kras, is possible.

2.3.2 OVPsim

gem5 provides cycle-accurate CPU models, i.e. CPU models that simulate a CPU one cycle at a time. This is useful for some research and design exploration, but comes at the cost of long execution times. It is certainly faster than RTL simulation, but sometimes not yet enough. A faster approach to CPU simulation is just-in-time (JIT) compilation of the simulated instructions to code that can run on the host. While this does result in less accurate timing information, simulations can run faster which has benefits in the functional evaluation phase [31]. A simulator that uses the JIT approach is OVPsim, a commercial product offered by Imperas Software [32].

Users do not need to create models from scratch. While the core of OVPsim (i.e. the simulator) is proprietary, many CPU models that use the OVPsim API are freely available. This includes support for ARM, x86 and MIPS architectures, some of them endorsed by the original manufacturer [33].

While it is not possible to run SystemC modules as part of an OVPsim simulation, the opposite is possible. Wrapping OVPsim models in SystemC TLM modules is supported out of the box. While this does introduce some overhead, users can still benefit from the fast OVPsim CPU models [34].

OVPsim is a simulation package that focuses on speed to enable quick functional analysis and development. As such, it is a valuable addition to the features already offered by gem5. Because support for SystemC TLM is part of OVPsim, integration into a framework like Kras should require little extra work.

2.3.3 GPGPU-Sim

The previous subsections analyzed software packages that provide models of CPU cores. While the CPU is at the center of a SoC, many components are still missing. One component often found on SoCs are graphic processors (GPUs). SoCs with built-in GPUs are produced for applications such as mobile phones and notebook computers [35]. A software package that simulates GPUs is GPGPU-Sim [36].

GPGPU-Sim is a freely available simulation library that can model various GPU configurations. Included in the current release are pre-configured setups for GTX480, QuadroFX5600 and TeslaC2050 processors. GPGPU-Sim can run programs developed for OpenCL and NVIDIA CUDA APIs. The current release of GPGPU-Sim is known to work with CUDA versions 2.3, 3.1 and 4.0 [36, 37], although the development branch (`dev`) of the project has support for more recent versions, including CUDA 8 [37].

Programs compiled to run on real CUDA hardware will work on GPGPU-Sim, simulation with GPGPU-Sim does not require re-compilation. Usually, programs that use CUDA APIs link against a dynamic runtime library provided by NVIDIA. This library provides the program with the required CUDA API functions which the library then redirects to the GPU driver. GPGPU-Sim provides a library of its own. It contains stubs that also provide the full CUDA runtime library. But instead of forwarding API requests to a real GPU driver, the API calls are handled by the GPGPU-Sim GPU model. Because GPGPU-Sim acts as a shared library, CUDA programs can be simulated without source code changes [36].

GPGPU-Sim does not offer a native SystemC TLM interface. Instead of directly coupling GPGPU-Sim with Kras, a wrapper is required. This wrapper forwards the API calls in simulation to GPGPU-Sim running on the host. Details are explained in chapter 4.

Chapter 3

Method

This chapter introduces the design of Kras and the rationale behind it. A semantic gap between SystemC source code and abstract configurations leads to an intermediate language. This intermediary needs to to capture the primitives that make a configuration while also being reasonably easy to translate to SystemC code. For this task, the established INI file format is chosen because INI is well-supported in many programming environments and can represent all required primitives. To transform a configuration to a running simulation, Kras translates the configuration to SystemC code. A regular C++ compiler can compile this code which results in a simulation binary. Finally, this chapter concludes with a discussion of the general software architecture of Kras, which is provided as a shared library. Both console interface and graphical editor use this library.

3.1 Primitives of Configurations

Kras wants to provide a facility for editing configurations. These configurations need to be stored in some kind of format, such that they can be archived and re-used. This leads to the following question: How does one describe configurations in a file that can be saved and used again later? As there are many possible formats to choose from, the question of what components a configuration file needs to be able to describe has to be answered first. To put it in different words, the question is: What are the required primitives to formulate a configuration?

3.1.1 Primitives Inherited From SystemC

SystemC TLM models are the base for this project. It is therefore useful to consider the primitives provided by SystemC. These primitives were already described in detail in section 2.2.

I. SystemC modules encapsulate models of hardware. Modules are C++ classes describing the behavior of the modeled component. Modules describe the *type* of instances.

II. To place a model into a simulation, an *instance* (an object) of a module is created in a top level file. To instantiate a module, certain constructor and template parameter might be required.

III. The interface of a module is described by its sockets. While SystemC differentiates between initiator and target, Kras simply refers to both of these connects as *bind points*.

IV. Instantiated modules are *bound* using their socket's `bind` method.

3.1.2 Bus Configurations as an Additional Primitive

Using this current set of primitives, it is possible to create instances of certain types and then connect these instances by connecting their bind points with each other. However, there is one primitive desirable for system simulation missing from this list of primitives: A facility for describing bus configurations.

Buses, referred to as "routers" in the SystemC reference manual [10, p. 420f], play a major role in SoC designs. After all, a bus connects the different components with each other, rarely are components connected directly with each other.

Routers in SystemC are based on memory addresses. When a bus is set up, components that offer services (slaves) connect to the bus at a certain starting address S for a given width w bytes. Components that wish to use services (masters) connect to the bus at no particular address. During simulation, routers in SystemC do two things: (1) Forwarding and (2) address translation. When a master accesses a memory address in the range $[S, S+w)$, the request gets (1) forwarded to the slave mapped at that address. During forwarding, the request address is normalized such that an initial request for address $S+x$ is (2) translated to a request for address x. Where a slave is mapped into the bus remains transparent to the slave because requests from masters are translated: From the point of view of the component offering a service, all requests are for addresses in the range $[0, w)$.

Bus systems can be written in SystemC TLM, indeed SystemC TLM is specifically designed to allow modeling of buses and MMIO-style interfaces [10, p. 413]. However, SystemC offers no abstract way to describe the routing setup inside a bus. Because bus configurations are such a fundamental part of SoC designs, the Kras configuration format has to offer *bind ranges* as an additional primitive.

V. *Bind ranges* act as bind points that multiple other bind points can connect to; at a certain address S for a given width w. Binding to a bind range means configuring the router.

In total, five primitives were identified. Types (item I.) define the properties of instances (item II.). In particular, they describe the available bind points (item III.) and bind ranges (item V.) that can be used to bind (item IV.) instances together.

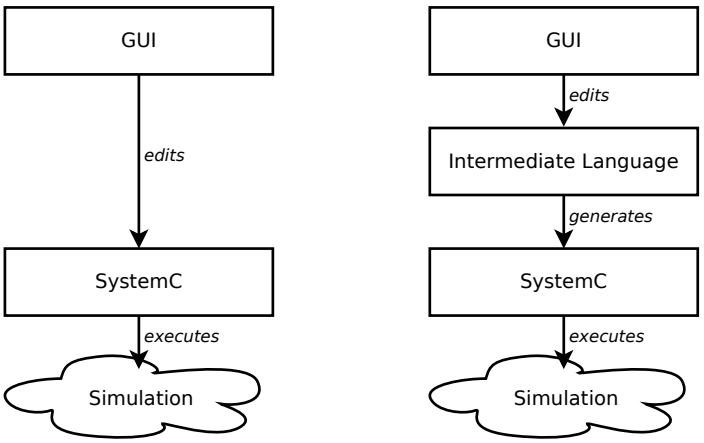
3.2 Configuration Format

To find a fitting file format for this task, it is helpful to consider the ways users are expected to interact with these configurations. For one, (1) users might edit the configuration through the graphical front-end. On the other hand, (2) users might edit the files themselves with a text editor. Both approaches are legitimate and should be supported.

3.2.1 SystemC

The models used by Kras are written in SystemC. It comes natural to then use SystemC source files as the medium for storing configurations. This approach offers advantages, in particular for use case (2) in which the user edits the configuration using a text editor. For one, this means that a user only has to know and use one language, SystemC, to both write models and set up configurations. A user already familiar with SystemC does not need to learn a new language. Another advantage is that because SystemC code is edited directly, there is no ambiguity about how statements in the configuration file translate to SystemC code. This stands in contrast to approaches that use a different configuration language, as this adds a translation step that might not be immediately understood.

While these are two advantages of SystemC as a configuration format, there is a laborious downside to this approach: SystemC code is actually C++ code. Unfortunately, the C++ programming language is notoriously hard to parse and understand [38] which makes it hard to load a configuration written in SystemC into a graphical editor. This problem is an example of a semantic gap [39]: SystemC code and graphical front-end have to encode the same configuration, but translation between those two representations means crossing the *gap* between them. In this case, the gap might be too big. that SystemC as a configuration language stands in contrast with the initial wish for abstract configurations. Configurations should be concise, something that might not be the case for configuration phrased in C++. While using SystemC as a configuration language would avoid the need for another language, the level of detail of C++ is too high. Because of this, SystemC as a configuration format was rejected.



(a) The SystemC configuration is edited directly.

(b) The configuration is stored in an intermediate language which is then turned into a running SystemC simulation.

Figure 3.1 – Directly modifying SystemC means less layers of abstraction, but results in a bigger semantic gap.

3.2.2 Intermediate Languages

To turn the semantic gap into a manageable one, an intermediate configuration format offers support. Conceptually, it sits between abstract idea represented by a graphical front-end and SystemC code. This is illustrated in Figure 3.1. This intermediate language needs to be able to (1) concisely describe configurations while (2) being reasonably easy to translate to SystemC. Two languages were considered: A custom domain specific language (DSL) and the established INI format.

In both cases, all primitives are represented in roughly the same way. At the center of a configuration are instances of a certain type and these instances have *mappings* associated with them. A mapping is a key/value pair that can be a constructor or template parameter or an active binding. All information specific to an instance is stored in its mappings. Kras stores type information for each key to associate the

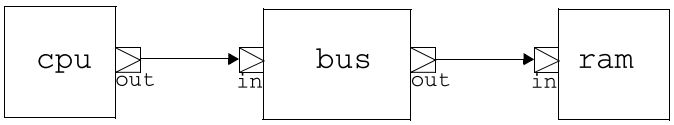


Figure 3.2 – An example setup. A CPU acts as a master and RAM as a slave. Both components connect to a central bus.

value with a constructor parameter, template parameter or binding. Only with type information can mappings be translated to the correct SystemC code.

To illustrate both alternatives, a basic configuration is described using either language. The setup is pictured in Figure 3.2 and consists of only three components, a CPU and some RAM, connected with a bus. While this is a simple design, it requires all five primitives listed in section 3.1.

3.2.3 Domain Specific Language

An approach that was considered was the design of a DSL. In general, a DSL is a language designed for one specific use (the problem domain). Because it is so limited in use, only required features are included. The language can be descriptive and focus on core components, omitting ceremony and unnecessary detail [40].

An example configuration of a suggested DSL for Kras configuration is presented in Listing 3.1. All required primitives are available. Each instance is started with the name of the instance, followed by the type, akin to representations in some programming languages [41, 42]. Inside curly braces, the different mappings describe the properties of the instance. The syntax for bind ranges is short and concise.

Using this DSL has advantages. Because the DSL can focus on the problem domain, there is no extra verbosity, the configuration is described in a succinct way. Properties like the type of an instance are easy to represent [40].

However, DSLs also come at a price. Because for every domain a new language is created, no library support can exist. Programmers have to write a new parser or generator to support the format. Also, the performance advantages of well-maintained existing configuration language libraries should not be underestimated [40].

```
1  cpu: CPU {
2    threads: 2
3  }
4
5  bus: Bus {
6    in: cpu.out
7    out ram.in[0x1992,0x10]
8  }
9
10 ram: RAM {
11   capacity: "4G"
12 }
```

Listing 3.1 – A possible DSL configuration.

```
1  [cpu]
2  type=CPU
3  threads=2
4
5  [bus]
6  type=Bus
7  in=cpu.out
8  out=ram.in@0x1992,x010
9
10 [ram]
11 type=RAM
12 capacity="4G"
```

Listing 3.2 – An INI confiugraiton.

3.2.4 INI

Another considered format is the INI file format, a general serialization format. While INI is not a formal standard, it has seen wide adoption in many MS-DOS and early Windows applications [43, p.803] and remains in use today, e.g. in free software projects [29, 44].

Listing 3.2 illustrates how configurations are described using INI. *Sections* represent instances, initiated by `[sectionname]` where the name of the section is also the name of the instance. An obligatory `type` is required to indicate the type of the instance. For representing bind ranges, a special format for the value was chosen, similar to the suggested DSL, but with a syntax that does not required any square brackets, as these are already in use in INI.

Representing configurations as INI files results in compact files. Little boilerplate is required with the only downside being the required `type` field. Because INI libraries are available for many programming languages [45–47], reading and creating INI files is quickly implemented. The only downside is the way ranged mappings are implemented. They do require some extra logic, but compared to implementing a whole language, this is a small problem.

Both approaches are viable, that is they can describe the required primitives. A DSL means no boilerplate and unnecessary features, but it adds additional work to programmers wanting to implement software that uses this format. INI is an existing format with library support that remains concise. The final version of Kras uses INI files to represent configurations.

3.3 From Configuration to Running Simulation

At this point, there is a way to formulate configurations as INI files. The next step is translating this abstract representation into an actually running simulation that

produces diagnostic output for evaluation and experimentation. Because Kras uses SystemC to actually run the simulation the next problem is the following: How is it possible to translate an abstract configuration file into SystemC TLM objects and bindings, ready for simulation? For this, two approaches were considered. The first one, *dynamic setup*, aims to instantiate objects dynamically during runtime. While the dynamic setup offers flexibility in theory, it is ultimately not compatible with SystemC. Because the first approach fails, *code generation* is chosen as a viable alternative. Configuration files are translated to SystemC code which is then compiled and run using a regular C++ tool chain.

3.3.1 Dynamic Setup

Initially, Kras was supposed to load configuration files dynamically, Figure 3.3 illustrates this idea. Kras would provide a program *krassim* that includes all models and peripherals and the required SystemC code for executing these models. The configuration file would be loaded by this program, which would then dynamically instantiate SystemC TLM modules (using the `new` operator), call the required `bind` methods to configure the sockets and then start the simulation.

This approach was considered because it should offer some advantageous features. For one thing, this approach should make it easy to deploy Kras. All that is required to run a simulation is a configuration file and the krassim program. In particular, no SystemC or other headers need to be found and included by the end user. It is a user experience similar to emulators like QEMU where the user configures the system in a configuration files or through command line arguments [48]. Another advantage is that it should be relatively fast compared to more involved alternatives such as code generation. Setting up the SystemC models and bindings in memory is fast compared to laborious compilation. Overall, this approach was considered because it should offer an easy to use interface and good performance.

While these are some worthwhile advantages, implementation of such a system is simply not possible with SystemC. This is because SystemC heavily relies on C++ templates [10, p. 426f]. Many SystemC modules require more than just constructor parameters for instantiating them, they also require template parameters. As an example, all initiator and target sockets have a template parameter `BUSWIDTH` [10,

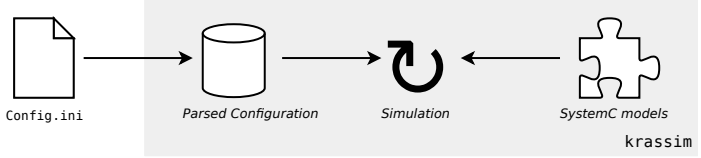


Figure 3.3 – Dynamic Setup. All steps take place in the krassim process (gray). krassim combines the configuration (parsed from an INI file) and SystemC models (pre-loaded on startup).

p. 456f] that defines the number of bits the modeled socket uses for communication. In practice, that means that there is not just one kind of initiator or target socket, rather there are many different *candidates*, all with a different `BUSWIDTH`. Template resolution happens at compile time and only those candidates get included in the binary that are used in the program [49, p. 21ff]. To consider the hypothetical krassim program, it would have to know in advance what values for `BUSWIDTH` are required. While it is certainly possible to limit this number to a few reasonable choices (e.g. 8, 16, 32, etc.), this is (1) limiting regardless and (2) there are many more template parameters which do not have immediately apparent values. Even if they did, the size of krassim would grow exponentially with the number of template parameters. Even worse, users would be limited to the models already included in krassim. Adding or modifying a model would require rebuilding the entire application. While at first glance the dynamic setup is more user-friendly, it is actually impossible to implement and even if it were possible, very cumbersome for the user. Because of these reasons, the dynamic setup approach was discarded.

3.3.2 Code Generation

The dynamic approach failed. As such, a different way to translate configuration files into running simulations had to be found: The INI configuration is translated to SystemC source code. A regular SystemC/C++ compiler then compiles this code to a simulation executable. The gist of this approach is illustrated in Figure 3.4.

For starters, this approach is actually implementable: Four out of the five primitives used in configuration files stem from SystemC, as such it is trivial to map these primitives to SystemC. The additional primitive, bind ranges (item V.), can be implemented as well: SystemC implementations of buses have to provide a special method for configuring their bus configuration. During code generation, Kras calls

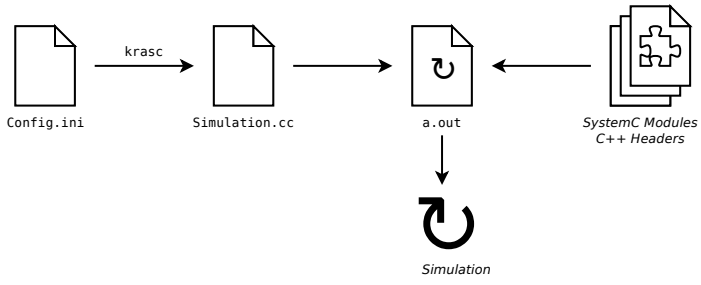


Figure 3.4 – Code Generation. The krasc utility translates the INI file into a regular C++ program that uses SystemC modules described in C++ headers. Compiling the program and running the binary (`a.out`) means running the simulation.

this method to configure the bus during elaboration of the SystemC configuration. The details are described in subsection 4.1.4.

In particular, templates are not a problem for code generation. Kras generates regular C++ object instantiations with all required template parameters. The additional compilation step from SystemC source files to binary then creates the specialized versions (candidates) of the used SystemC modules. Because regular C++ code that uses templates as intended is used, code generation is an approach to loading simulations from configuration files that can actually work.

Code generation in favor of a dynamic setup also makes Kras more extensible: Adding new models or modifying existing ones does not require re-compilation of the Kras tool chain, only of the final simulation binary. On the downside, code generation complicates the simulation workflow. Where before one could simply supply a program with a configuration file, now more complex setups are necessary. To achieve reasonable productivity, this workflow will require scripting or use of build systems. However, this can also be seen as an advantage: Because Kras now takes up a smaller portion of the simulation workflow, the system becomes more cooperative with others.

The initial design of Kras was that of a single binary that would load configuration files and then run the simulation. Because of heavy use of C++ templates in SystemC and a complicated upgrade path requiring re-compilation of the Kras tool chain, this option was discarded in favor of code generation. Kras generates SystemC code from configuration files. Compiling this generated code results in a SystemC simulation.

3.4 Implementation as a Shared Library

So far, this chapter was occupied with models and their simulation. This section explains the general software architecture of Kras and the rationale behind it.

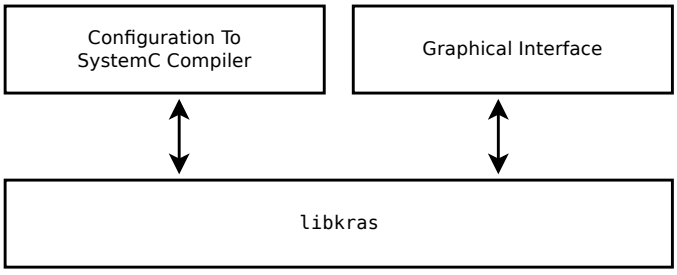


Figure 3.5 – Rough software architecture of Kras. At the core is a shared library, libkras, which provides functionality such as parsing and editing configuration files as well as generating SystemC code. The applications that sit on top of the library are only front-ends to the features available in the library.

In general, Kras wants to offer a facility for editing simulation configurations, including a graphical editor. These configurations then need to be run, which will be implemented by translating configuration files to SystemC code. From this, two main interactions a user has with Kras can be made out: (1) Editing configuration files and (2) generating SystemC code from configurations. Considering the first use-case, (1) editing configuration files, this can be achieved both by editing the text file itself or by using the graphical editor. While the first option requires no additional support from Kras, the graphical interface certainly will. To display a configuration, a configuration file needs to be loaded, the individual instances and their connections need to be identified. Looking at the second interaction, (2) generating SystemC code from configuration files, the requirements are similar to (1). Just like before, to generate a SystemC file, first the configuration file needs to be loaded and understood. Both user interactions have similar requirements, in both cases the configuration format always needs to be parsed into some kind of in-memory representation to work with.

Motivated by this, the architecture of Kras is as follows: At the base is a shared library, *libkras*. It can load an INI configuration and return an in-memory representation of that file, similar to how *abstract syntax trees* represent code fragments in compilers [50, p. 41f]. This abstract representation can then be displayed by the graphical editor or used to generate the appropriate SystemC code. This general architecture is depicted in Figure 3.5.

Chapter 4

Implementation

The previous chapter described the general design behind Kras. This chapter now follows up by documenting the implementation created for this thesis. Kras is split in three parts: (1) A Library called libkras, (2) a command line tool *krasc* and a (3) graphical user interface called *Gras*. libkras offers an object-oriented API for loading, creating and modifying configurations as well as generating SystemC code. krasc is a compiler that translates INI configurations to SystemC. Because it is a simple command line tool is is easy to integrate krasc into build setups. The Gras tool is a graphical editor for configuration files. Both command line tool and graphical interface are not very useful without models to configure, so the existing simulation packages introduced in chapter 2 were adapted to work with libkras.

4.1 Core Library

The core functionality of Kras is implemented in the form a shared library, libkras. It offers an object-oriented API to applications for setting up configurations. Configurations are created either from loading INI files or from scratch and are represented by a tree of objects in memory (the *in memory configuration*). Using API calls on objects from this tree, the configuration can be modified as desired. After editing is complete, libkras can write the configuration back to an INI file for later use or translate the configuration to runnable SystemC code.

The library is implemented in C++. Initial experimentation with the dynamic setup approach are the first reason for this. The dynamic approach certainly required a C++ code base such that SystemC models and simulation could be initialized. While the dynamic setup approach did fail, many parts of the current implementation were salvaged from this first attempt. C++ isn't necessarily an ideal choice. The language is not as productive and more prone to defects compared to a high level language such as Java [51]. The Kras project also does not require the performance

advantages of a low-level language such as C++ as it does not run any time consuming algorithms. However, picking C++ still makes sense considering all the connections with SystemC during development. In particular, because all models are already written in SystemC and therefore C++, it means that the entire project only uses one programming language instead of multiple different ones.

4.1.1 Object Oriented Configuration

To understand the libkras library, it is necessary to take a look at how configurations are represented using the object-oriented interface.

Essentially, applications that use libkras work on a tree of objects which represents a configuration. As an overview, Figure 4.1 illustrates the involved classes and their relationships. At the root of a configuration is a `Config` object. An application is free to create a new empty configuration from scratch or load one from an existing INI file. An instantiated model is represented by an `Instance`. To create an `Instance`, it requires a `name` unique to its `Config` and a `type` which is the model that should be instantiated. The following subsection 4.1.2 explains how types are represented in libkras; for now it is sufficient to understand that type information is stored together with instances. `Config` and `Instance` are enough to instantiate C++ classes in the final simulation if they have an empty constructor, i.e. classes that do not require any template or constructor parameters.

What is missing is a way to create instances that require template or constructor parameters. In addition to this, a way to represent bindings is still missing as well. Such configuration details are stored as *mappings*, similar to how they are represented in INI files. A `Mapping` can be understood as a single key/value pair. It is used to configure the `Instance` it is associated with, that is it describes the

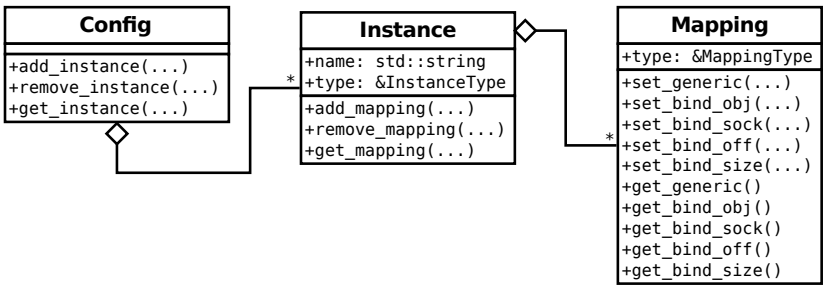


Figure 4.1 – Simplified class diagram showing the three main classes involved in a configuration. A `Config` makes up the root of a configuration. This configuration then contains multiple `Instances`. These instances are configured with key/value mappings represented by `Mapping` objects.

properties of a concrete instantiation. Depending on the value of `Mapping.type`, such a mapping may represent one of the following:

- A template parameter, required for instantiation.
- A constructor parameter, required for instantiation.
- A binding to a bind point, i.e. a point to point connection between two instances.
- A binding to a bind range, including the address offset S and width w.

Depending on the `type` of a `Mapping`, different accessors have to be used. That explains the many getters and setters in the class diagram for `Mapping` (Figure 4.1). To illustrate this with an example, a mapping representing a constructor parameter will only support `Mapping::get_generic` which returns a string. On the other hand, a mapping representing a bind range supports `Mapping::get_size` (among others) to return the width of the mapping. As a consequence, applications have to check the `type` before accessing the individual values. If an application accesses an inappropriate getter or setter, the library throws an exception.

A maybe more elegant implementation is possible. Instead of only one class to represent all mappings, multiple specializations (`ConstructorParameterMapping`, `BindPointMapping`, etc.) would inherit from an abstract base class. While this is more in line with the ideas of object-orientation, an application would still have to check the type of a given mapping. As such, the simpler approach that uses only one `Mapping` class was chosen.

All mapping information, including bindings, is stored as strings, rather than pointers to other instances. This allows applications to dynamically create designs without worrying about compatibility with the current state. On the downside, simply using strings also results in a new problem: Bindings are referenced by string, i.e. by the (1) name of the neighboring instance and the (2) name of the neighbor's bind point. Both of these names might not actually exist in the current configuration, which results in SystemC that does not compile. To avoid C++ compile errors late in the workflow, error checking was added to the library. libkras offers functions that search through configuration and ensure that all referenced instances and bind points actually exist.

The three classes that represent configurations were kept simple. They act as mere containers: Aside from storing configurations, they do not offer any features. Additional functionality, such as error checking or code generation, is provided by library functions that read an existing `Config`. Consequently, the core data structure representing a configuration can be kept simple and as such easy to manipulate and understand.

4.1.2 Type Information

The previous section described how configurations can be arranged in memory using the object oriented API. What was omitted were the origins and structure of types, i.e. where type information comes from and how it is represented in libkras. Type information is required so libkras knows which mappings correspond to constructor or template parameters and which one to bindings; without type information, SystemC code could not be generated. Type information is also useful for interactive editors: If the library knows what mappings are available for a given type, the tool can recommend them to the user or offer auto-complete functionality.

Both instances and mappings have types. Each `Instance` has an `InstanceType` associated with it, in turn each `Mapping` is classified by a `MappingType`. Types are managed by a central authority, the `TypeRegistry`. To illustrate the interaction between the different classes, Figure 4.2 shows their relationships.

All types are organized in a `TypeRegistry` which acts as a collection for all instance types. A challenge during development was finding a way to fill this `TypeRegistry` with type information collected from SystemC modules, i.e. how to extract the required type information (name, template parameters, constructor parameters, bind points, bindings) from SystemC modules. Ideally, this would happen automatically: Kras should read the SystemC files and extract the required type information. This might be doable using the LLVM and clang libraries [52, 53], but because of time constrains this approach could not be followed up upon.

Instead of an automatic solution, users have to do some extra work to make SystemC modules ready for use with Kras. For the type information of SystemC modules to be loadable by a `TypeRegistry`, the SystemC files need to be *annotated* by the user. To allow for flexible setups, two kinds of annotations are currently supported, *internal annotations* and *external annotations*.

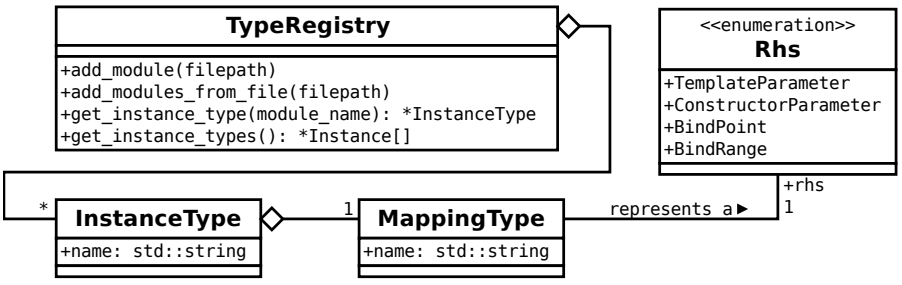


Figure 4.2 – Simplified class diagram showing the three main classes used for type analysis of configurations. `TypeRegistry` acts as a central authority for types. An `Instance` is associated to an `InstanceType` which contains `MappingType` objects for all available mappings.

```
20 #define HAS_KRAS_TEMPLATE_PARAM(NAME) /* noop */
21 #define HAS_KRAS_CONSTRUCTOR_PARAM(NAME) /* noop */
22 #define HAS_KRAS_BIND_POINT(NAME) /* noop */
23 #define HAS_KRAS_BIND_RANGE(NAME) /* noop */
```

Listing 4.1 – These macros can be used in SystemC modules. They have no effect in SystemC itself, but are picked up by the `TypeRegistry`. Exerpt from `krasmodules.hh`.

Internal annotations are put directly in the SystemC file defining a module. To do this, SystemC modules include the `krasmodules.hh` header (Listing 4.1). It defines various macros that have no meaning in the SystemC language itself, but are recognized by libkras. An annotated file can be loaded into the registry using the `TypeRegistry::add_module` method. The file is loaded and the macros identified using regular expressions. The advantage of this approach is that SystemC model and annotation are united in one file. While it does mean code duplication, at least the duplicated information is grouped in the same file. A limitation to consider here is that the current tooling assumes that one SystemC file corresponds to exactly one SystemC module named after the name of the file. This is not always the case.

Internal annotations are limited and sometimes it isn't possible or desirable to change the contents of a SystemC module, in particular when using modules provided by library code. To overcome this problem, external annotations offer an alternative. External annotations reside in separate INI files, an example is demonstrated in Listing 4.2. Invoking the `TypeRegistry::add_modules_from_file` method reads the selected INI file and parses out the required type information. While this means that the type information is scattered in two files, once in the actual model and once in the annotation file, it is a solution to the problem as existing SystemC code does not require change.

Type information is required to check the validity of configuration files, offer auto-complete functionality and generate SystemC code. Type information is collected

```
1 [Printer]
2 ConstructorParameter=name
3 BindPoint=in
4
5 [Sender]
6 ConstructorParameter=name
7 ConstructorParameter=dest_addr
8 BindPoint=out
```

Listing 4.2 – An example external annotation file. It describes two types, `Printer` and `Sender`.

either from internal annotations in the SystemC files themselves or from external annotation files and organized in a `TypeRegistry`.

4.1.3 Parsing INI files

libkras uses INI files to store configurations. INI is an established format and as such many libraries that parse and generate INI files are available. The library libkras uses to parse INI files is the versatile *property tree* class provided by the Boost project [47]. A property tree acts as a container for key/value pairs where values can again be key/value pairs. Boost includes property tree parsers and generators for many established configuration formats, including INI and JSON. Using the property tree in libkras is straight-forward: The Boost library parses the INI file which returns a tree. libkras iterates on that tree and builds a matching `Config`.

The boost library was chosen because of two main reasons. For one thing, (1) boost is widely available and released under a liberal license [54]. The second reason is (2) support for other serialization formats. The property tree library also supports reading and writing formats such as JSON and XML. Should there ever be interest in switching Kras to a more advanced configuration format, e.g. because a new feature cannot be represented with the simple INI format, it should prove to be relatively easy to port to JSON or XML.

4.1.4 Translation to SystemC

The ultimate destination of a Kras configuration remains simulation. To achieve this, libkras generates SystemC code from the in-memory configuration. All components of a Kras configuration need to be mapped to SystemC representations.

As previously explained (subsection 2.2.2), SystemC simulations work in two phases: Elaboration and actual simulation. Elaboration sets up instances and their bindings. Simulation supplies the instances with stimuli and records their responses. Kras only needs to occupy itself with elaboration, actual simulation is handled by SystemC. As such, what libkras needs to generate is the elaboration of a simulation.

Mapping instantiation, type and `bind` is trivial. After all, they are SystemC primitives and as such have a direct mapping to SystemC. However, bind ranges are not part of the SystemC standard. The standard only knows of initiator and target sockets, grouped together as bind points in Kras. How can bind ranges, that is bus configurations, be translated to SystemC?

Connecting initiator to target is done with a call to `bind` on one of the sockets. This is possible because all SystemC TLM sockets inherit from a base class that provides the `bind` method. Inspired by this, bind ranges are implemented in a similar fashion. First, a C++ base class, `IBindRange` (Listing 4.3), is included. SystemC modules that wish to implement bind ranges, e.g. classes that act as

```
31 /*
32  * SystemC modules that want to offer Kras BindRange ↘
        functionality
33  * should inherit from this class.
34  */
35 namespace Kras
36 {
37   template <unsigned BUSWIDTH = 32>
38   class IBindRange
39   {
40   public:
41     virtual ~IBindRange() = default;
42     virtual void bind_at(tlm::tlm_target_socket<BUSWIDTH> &s, ↘
          uint64_t off, uint64_t size) = 0;
43   };
44 }
```

Listing 4.3 – The `IBindRange` class offers the `bind_at` method. Exerpt from `krasmodules.hh`.

buses or "routers" [10, p. 420f], inherit from this class. Primarily, `IBindRange` contains one virtual method, `bind_at`, which is provided analogous to `bind`. Classes that inherit form `IBindRange` have to implement this method such that the target socket `s` will be accessible at address `off` for a total of `size` bytes. Compared to previous discussion (subsection 3.1.2), `off` is analogous to start address S and `size` is analogous to width w. Note that `s` may only be a target socket. To connect an initiator socket to the bus, the bus will have to provide at least one initiator socket of its own, but that is up to the implementation of the bus module. In short, buses have to inherit from the `IBindRange` class and provide the `bind_at` method. libkras translate bind range connections to `bind_at` invocations.

Now with all primitives accounted for, elaboration code can be generated. In libkras, SystemC elaboration is split in two phases: First, (1) instantiation which creates the individual objects and second, (2) binding, which connects the individual objects with each other. This split simplifies code generation. The (1) first step creates individual objects which are totally independent from each other. It does not matter in which order libkras creates these instances, any order is semantically identical. Therefore it is sufficient to iterate over all available instances and emit an object instantiation. The (2) second step creates dependencies between instances (both need to exist for a binding to be possible), but the individual bindings themselves are independent from each other. Again, SystemC translation is easy: For each binding of every instance, libkras emits a `bind` or `bind_at` call. This shows that code generation is made easy by splitting up the process into multiple steps.

4.2 Command Line Interface

With the library in place, implementation of command line tools is trivial. Provided with Kras are two command line utilities.

- The krasc program translates INI configurations to SystemC sources. First, depending on its arguments, krasc loads type annotations from an arbitrary number of SystemC modules and external annotation files to create a `TypeRegistry`. With the type information loaded, krasc parses the the INI configuration file and generates matching SystemC code.

  All hard work is handled by the libkras library. The only contribution of krasc is that the generated code can be inserted in a prepared template file.

- Additionally, the *krasmodinfo* tool is provided. It reads internal or external annotations and prints them in the expected format for external annotations. krasmodinfo is only provided for debugging annotations and again delegates all work to the library.

4.3 Graphical User Interface

Compared to building the command line tools, implementing the graphical user interface Gras required more work. This stems from the fact that building a graphical editor is more involved than calling library functions: Data structures of the library need to be displayed and be editable using mouse and keyboard.

During design of Gras, the question was how to represent configurations in a meaningful manner. Previous work on graphical editing of SystemC RTL models [55, 56] and the user interface of the big virtual platforms (Platform Architect, Vista) suggest a schematic overview of the configuration; individual instances are represented as rectangles with their connections indicated by lines between them. This approach was taken up for Gras as well. Figure 4.3 shows the main interface of Gras: Instances can be dropped in, arranged and connected with the mouse.

While there remains lots of room for additional features, e.g. a source code viewer [56] or automatic integration in build systems, the set goal for Gras was to create a functional prototype that can display configurations as a schematic and allows users to edit and save these configurations.

Gras is implemented with the Qt framework. Qt is a set of C++ libraries that can be used to build rich graphical applications. In fact, Qt contains more than just a framework for building graphical applications: It includes commonly required data structures such as strings, lists and hash maps [57]. Because Qt API works primarily with Qt data structures, Gras uses the Qt data structures in favor of the

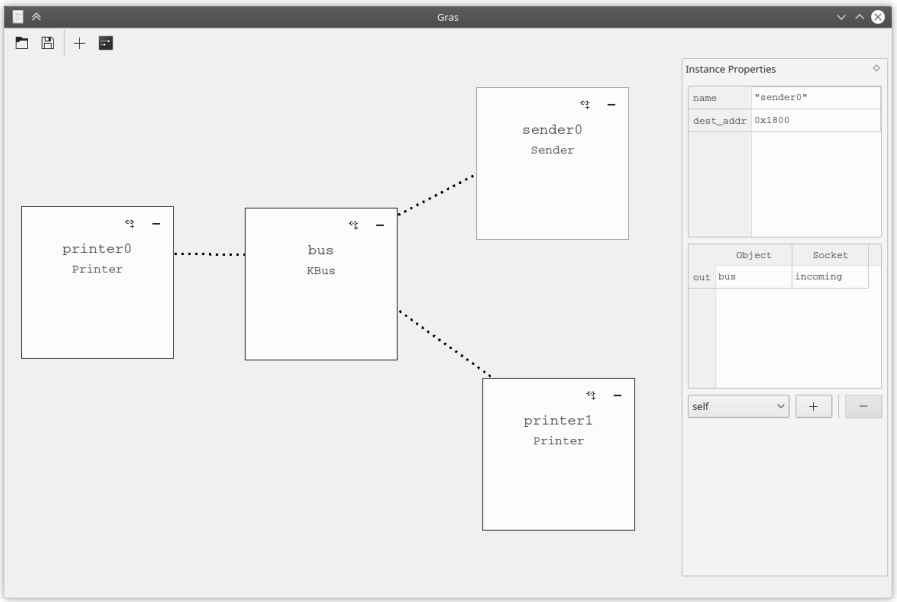


Figure 4.3 – Gras editor showing a basic setup. Individual instances are represented by rectangles, bindings by a line between instances.

C++ standard template library (STL) [58] alternatives, which are used in libkras. This does introduce some boilerplate code and runtime cost as data needs to be constantly translated between STL and Qt data structures, but in practice this proved to be insignificant.

When the GUI is started, it creates an empty `Config` object or loads one from an existing INI file. The graphical representation of the configuration is then adapted to fit the state of the `Config`. When the user clicks on an instance widget, a table on the right (Figure 4.4, the *mapping table*) can be used to view or edit the individual mappings of that instance. This introduces a problem: There are now two representations of the configuration, once inside the library code (wrapped in a `Config` object) and again in the state of the graphical widgets.

A challenge during development was how to keep the contents of the mapping table in sync with the in-memory configuration. Qt provides a streamlined solution to solve this issue in the form of the *model view* approach [59]. The idea behind this approach is the following: Provided a data source and a widget that displays that data (the view, e.g. a table in the GUI), an intermediate component, the model, handles interaction between data and view. In the projects' specific case, the data is provided by an `Instance` and the view is a table on the sidebar. In between sits the model, implemented as a C++ class, that on one hand interacts with the data source using its API (the libkras API) and on the other hand can interact with the view. Callback functions in the model get called when data is entered or changed in the

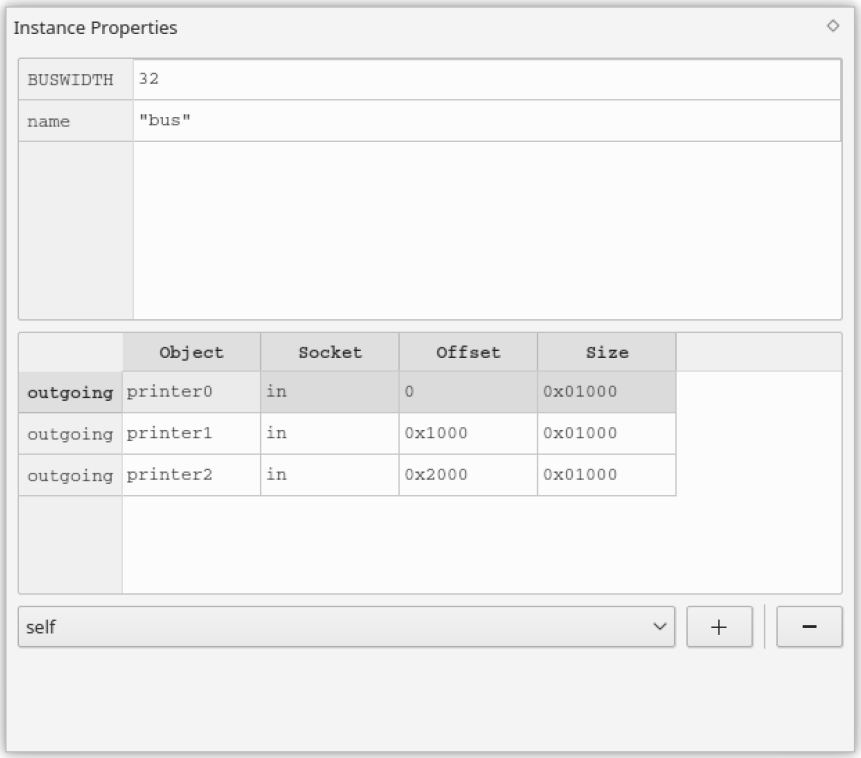


Figure 4.4 – Gras mapping table showing the individual mappings of the selected instance.

table, triggering a change in the underlying `Instance`. This way, the underlying in-memory representation always stays in sync with what is displayed in the graphical interface.

4.4 Porting SystemC Modules to Kras

The chapter on fundamentals gave an overview over three simulation packages, gem5, OVPsim and GPGPU-Sim. To integrate them into Kras, these components need to be accessible as SystemC modules annotated for use in Kras. To put it in more concrete terms, all simulators should be represented as a SystemC module that is accessible through bind points.

4.4.1 gem5

Recent work added a bridge between SystemC and gem5 [30]. Similar to how SystemC TLM uses initiator and target sockets, gem5 uses *master ports* and *slave ports*

for communication. A compatibility layer bridges SystemC sockets and gem5 ports with *transactors*. A transactor connects a SystemC initiator to a gem5 slave or vice versa a gem5 master to a SystemC target.

To instantiate a gem5 CPU in a Kras configuration, two SystemC modules provided as part of the gem5 compatibility layer are sufficient.

1. `Gem5SimControl` is a SystemC module that acts as a central instance for gem5, i.e. the simulated gem5 cores. A SystemC configuration that wishes to use gem5 models has to instantiate exactly one `Gem5SimControl`.

   The `Gem5SimControl` module requires a configuration file during instantiation. Users usually set up gem5 using a Python program, gem5 executes the Python code which results in an INI file. This INI file contains the full setup of the gem5 configuration; to set up a `Gem5SimControl`, such an INI file is required.

   To get this INI file for use in SystemC, users can write a basic Python configuration file and then have the gem5 binary translate it to INI. The authors of the SystemC bridge encourage future work on automating or finding ways to omit this step, but currently there is no working solution [30].

2. For SystemC modules to interact with the gem5 cores, a transactor is necessary. The `Gem5SlaveTransactor` offers that functionality. This transactor has one target socket `sim_control` and one initiator socket `socket`.

   Connecting `sim_control` to the singleton `Gem5SimControl` and `socket` to a bus or peripheral is enough to expose the CPU to SystemC.

Both of these modules, `Gem5SimControl` and `Gem5SlaveTransactor`, are provided by the gem5 project for use in SystemC configurations. Because they are provided as library code, external annotations provide Kras type information for both modules.

4.4.2 OVPsim

OVPsim is advertised as a simulation package that supports SystemC out of the box [32]. Indeed, an installation of OVPsim contains various SystemC modules that represent OVP CPU models.

For example, the `processor.igen.hpp`[1] header includes a SystemC module `arm` which represents an ARM CPU. Configuration of this `arm` module is simple, it only requires a configuration object which contains the *variant* of ARM core to simulate, e.g. Cortex-A57MPx4, as a string. For communication with other SystemC modules, `arm` provides two initiator sockets, `DATA` and `INSTRUCTION`; a configuration binds

[1] `ImperasLib/source/arm.ovpworld.org/processor/arm/1.0/tlm2.0/processor.igen.hpp` in Version 20170511.

them to data and instruction memory. If both data and instruction memory are the same, i.e. refer to the same instance, it's easy to write a wrapper for `arm` that forwards requests from both sockets to a single RAM or bus connect.

To use OVPsim modules in Kras, annotation is necessary. If a configuration contains the SystemC modules as-is, external annotation is the right choice. However, if a user decides to write a wrapper (e.g. to combine the `DATA` and `INSTRUCTION` sockets), internal annotations should be preferred.

4.4.3 GPGPU-Sim

GPGPU-Sim does not support SystemC TLM out of the box. As such, adding support is more involved, but possible.

First, a refresher on how GPGPU-Sim is usually used. From the point of view of an application, GPGPU-Sim provides the interface of a real graphics API, such as the CUDA API. This is achieved with the *runtime library*: The application links against this library which then forwards all request to the GPU model. As such, GPU simulation occurs on the machine that is also running the application using the runtime library. This is undesirable in the case of simulation: When a Kras configuration includes a GPU, GPU simulation should occur on the host machine instead of on the simulated gem5 or OVPsim guest. To solve this, an additional layer of abstraction is necessary.

Figure 4.5 illustrates how guest code can interact with a GPGPU-Sim GPU model. A SystemC module presenting the GPU to the simulated system gets added to the configuration. Guest code can access this GPU using MMIO registers to issue CUDA API calls. What registers to read and write with which values is wrapped in a *guest runtime library*, a library the guest code links against. This guest runtime library offers the full CUDA API to simulated programs, such that simulation remains transparent to guests. Once an API call has reached the SystemC GPU, it uses the

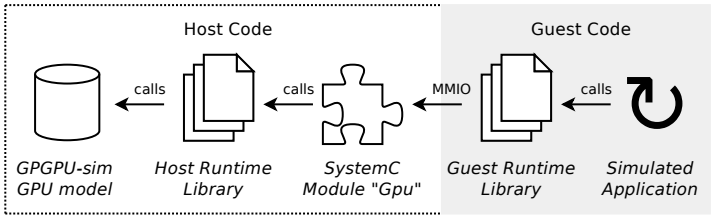


Figure 4.5 – Communication between guest code and simulated GPU. The guest code uses its own runtime library to talk to a SystemC module which represents the GPU to the guest. This SystemC module uses the GPGPU-Sim CUDA runtime library to forward API calls to the GPGPU-Sim GPU model. As such, GPU simulation can run on the host.

GPGPU-Sim runtime library, the *host runtime library*, to forward the API call to the GPGPU-Sim model.

With these components in place, GPGPU-Sim can be integrated in SystemC. In total, only one SystemC module needs to be written. Being a new component for Kras, internal annotation is the reasonable choice.

Chapter 5

Evaluation

As a final step, the implementation had to be evaluated. This chapter is structured in two sections. First, (1) the evaluation investigates the overhead introduced by SystemC coupling compared to running the simulators as-is. It becomes apparent that using existing simulators with Kras, or more specifically, in conjunction with SystemC modules, can introduce a serious overhead. Second, (2) the goals stated in chapter 1 are taken up again: The implementation of Kras does follow the ideas of these initial goals, however things could be improved further.

5.1 Cost of SystemC Coupling

Kras uses SystemC to combine various existing simulation packages to peripherals described in SystemC. If a memory module implemented in SystemC acts as the main memory of the simulated system, all memory requests of the CPU need to access that module. Instead of using an internal implementation provided by the simulation package, the CPU communicates with the SystemC memory implementation using SystemC sockets. Communication through sockets does introduce overhead as messages need to get marshaled on the sender side and then again un-marshaled by the receiver. Here, the question of evaluation was to what extend this overhead adds to simulation time.

5.1.1 Test Setup

For evaluation of SystemC overhead, a matching configuration was set up both with and without SystemC coupling configured by Kras. In either case, a CPU core (provided by gem5 or OVPsim) communicates through a bus with memory to run a given benchmark program. On one hand, bus and memory are implemented in SystemC. Here, all memory operations of the simulated CPU have to pass through

SystemC bus and memory. On the other hand, the setup without Kras (referred to as the *vanilla* setup), used bus and memory implementations provided by the simulation packages themselves. Comparing simulation times for both vanilla and non-vanilla setup should give insights into what to expect from SystemC coupling.

Regarding software, a total of four benchmark programs gave insight into simulator behavior. On one hand, there are three stand-alone benchmarks that implement a single algorithm, based on previous discussion in literature [60]. They differ in the expected amount of load on memory and therefore the bus and the SystemC connect as a whole.

- The *Ackermann* benchmark $A(m, n)$ calculates the result of the Ackermann function $A(m, n)$ [61]. Because Ackermann is recursive but not tail-recursive, computing values of A requires many recursive function calls for most m, n. Many recursive function calls translate to lots of work on stack memory, as such Ackermann should show heavy load on bus and memory.
- The *Sieve* benchmark $S(n)$ calculates the biggest prime number $p \leq n$. For this, the Sieve of Eratosthenes algorithm [62] is used which works on an array of size n. Load on memory was expected to be great as every iteration of the algorithm does work on the array.
- The *Monte Carlo* benchmark $\Pi(n)$ approximates the value of π by sampling n randomly generated points [63]. In the implementation used for evaluation, generating a random number and evaluating the result does not require a lot of memory accesses. In consequence, load on the bus was expected to be low.

In addition to these three algorithmic examples, a suite of benchmarks, Coremark [64], runs various kinds of benchmarks, including matrix multiplication, list operations and CRC calculation. It is convenient because the result is a condensed score (*iterations per second*) that is easy to compare.

5.1.2 Results

This section presents the measured numbers and the overhead introduced by SystemC coupling. Possible explanations and conclusions are then drawn in the section afterwards. Because of problems related to software licensing, the benchmarks for gem5 and OVP ran on separate machines. This is not a problem as this evaluation is interested in the performance differences of simulation with and without SystemC coupling. Here, the differences between gem5 and OVPsim are not relevant.

First, coupling with gem5 was evaluated. For this, machine based on a 3.4 GHz Intel i7-2600 CPU with 16 GiB of RAM and the Debian 9.5 operating system ran gem5 commit `21a691748`. gem5 includes not just one, but multiple CPU models of

	Coremark*	Ackermann**	Monte Carlo**	Sieve**
Timing Simple	0.3	70.5	41.1	27.1
Timing Simple w/ Kras	0.8	211.8	128.5	84.7
O3	0.6	109.9	57.7	30.2
O3 w/ Kras	0.1	993.4	396.3	447.1

Table 5.1 – Simulation using gem5 cores with and without Kras integration. For gem5, the benchmarks calculated $A(3,7)$, $\Pi(10^5)$ and $S(9 \times 10^4)$. *) Iterations per second. More is better. **) Seconds. Less is better.

	Coremark	Ackermann	Monte Carlo	Sieve
Timing Simple	2.7	3.0	3.1	3.1
O3	6.0	9.0	6.9	14.8

Table 5.2 – Overhead x introduced by Kras. The setup using SystemC coupling takes x-times as long as the vanilla version to run the same workload.

varying levels of accuracy. For this evaluation, two CPU models were chosen, the abstract but fast *Timing Simple* model and the more detailed but, in comparison, slower *O3* model. In either case, gem5 was configured to simulate an aarch64 instruction set architecture (ISA). The results of the different benchmarks are listed in Table 5.1. Table 5.2 lists the overhead Kras introduces. It is clear from these results that SystemC coupling introduces a notable overhead. In the case of the the Timing Simple CPU, the version that uses a SystemC memory and bus typically takes three times as long to run. If the more detailed O3 model is chosen, simulation may increase in an order of magnitude.

Next, OVPsim was pulled for evaluation as well. All benchmarks ran on a 2.83 GHz Intel Core 2 Quad Q95550 CPU with 4 GiB of RAM and the CentOS 7.4.1708 operating system, OVPsim release 20170511. OVPsim offers various CPU models and architectures. For evaluation, the `arm` SystemC module in `processor.igen.hpp` was configured to simulate the properties of a Cortex-A53MPx1 CPU which is based on the aarch64 ISA. As before, Table 5.3 lists the actual Coremark score and execution times while Table 5.4 shows the overhead introduced by Kras and SystemC coupling. In the case of OVPsim, the difference can only be described as dramatic. Using the

	Coremark*	Ackermann**	Monte Carlo**	Sieve**
OVPsim	1989	2	26	4
OVPsim w/ Kras	4	754	7387	5164

Table 5.3 – Simulation using OVPsim cores with and without Kras integration. For OVPsim, the benchmarks calculated $A(3,10)$, $\Pi(10^8)$ and $S(N/A)$. *) Iterations per second. More is better. **) Seconds. Less is better.

	Coremark	Ackermann	Monte Carlo	Sieve
OVPsim	798	377	284	1291

Table 5.4 – Overhead x introduced by Kras. The setup using SystemC coupling takes x-times as long as the vanilla version to run the same workload.

version with SystemC bus and memory resulted in up to three orders of magnitude worse performance.

5.1.3 Origin of Overhead

Kras, or rather, SystemC coupling, introduces an overhead. While in the case of the generally slower gem5 cores, an overhead between three to 15 times the vanilla runtime can be observed, in the case of OVPsim the difference is even greater. This section investigates the origins of this overhead.

Before starting the actual discussion, it should be noted that the gem5 results are somewhat unexpected. Running a fast Timing Simple CPU with SystemC coupling takes about three times as long as without it, at the same time, running the slower O3 model with SystemC can add cost of an order of magnitude. The penalty of SystemC coupling should be higher if running SystemC code makes up a bigger percentage of the total runtime. This should be the case for the Timing Simple model as the CPU model itself spends less time than the O3 model to calculate each step of the CPU.

With this oddity in mind, the simulations were profiled to find out exactly where the programs spend their execution time. This should give insight into who is to blame for the reduced runtime of non-vanilla simulations. Two profilers were employed for this, *Callgrind* and *gprof*.

5.1.3.1 Overhead for gem5 Configurations

Callgrind [65] profiles a running program and counts the number of function calls to each function. With Callgrind, it is possible to estimate how much time the simulation binary spends in simulator code, the SystemC library or the code implementing bus and RAM peripherals. Table 5.5 shows the result for a run of the Ackermann benchmark. On one hand, these results are as expected: Using the more detailed and therefore slower O3 model leads to SystemC and peripheral code making up less of the total function calls, that is the more expensive simulation takes up most of the total execution time. On the other hand, this remains in contradiction to the results in Table 5.2, where the relative overhead of SystemC coupling is greater in the case of the O3 CPU. Notably, the O3 version calls way more C/C++ standard library functions, in particular memory management functions.

	gem5	SystemC library	Peripherals	C/C++ stdlib
Timing Simple	32.7	26.6	17.2	19.3
O3	52.8	3.8	5.7	37.1

Table 5.5 – Percentage of function calls into gem5 code, SystemC library code or peripheral logic (bus and memory) when computing $A(0,2)$. The numbers do not add up to 100% as this table omits some standard runtime functions.

Because Callgrind only counts the number of function calls, but not the time spent in a function, the gprof profiler [66, 67] was also used to profile the generated simulation binaries. It measures the amount of time each function takes to run. The results of gprof line up with the results of Callgrind: The O3 simulation spends less time in SystemC and peripheral code than the Timing Simple code.

It remains unclear why O3 simulations have such a big performance penalty compared to Timing Simple simulations. The most likely result from the traces leads to the non-vanilla O3 version spending a lot of time in memory allocation routines. Another possibility is that the gem5 cores are configured differently when running vanilla or together with SystemC modules. This could be attributed to the, admittedly now thoroughly checked, test setup or because of a bug in gem5.

The take-away from these results is that in the case of gem5, impact of SystemC coupling depends on the type of CPU model, not so much the performance of the SystemC peripherals. Regardless, overhead remains in levels that might be acceptable for many applications.

5.1.3.2 Overhead for OVPsim Configurations

To find out where the Kras setup using OVPsim spends most of its time, the resulting binary was benchmarked with gprof as well. Looking at all functions that make up at least 0.01% of execution time shows that at least 85% of execution time is spent with SystemC library code or the SystemC peripherals (bus, memory) used for the benchmarks. A lot of time is also spent in memory allocation routines related to `std::vector`, likely caused by SystemC coupling code. The conclusion must be that the dramatic overhead for OVPsim simulation is caused almost entirely by SystemC coupling and peripherals. Here, overhead could be reduced by writing more efficient SystemC modules, but because the coupling alone produces much overhead, the problem is not completely negatable.

5.1.4 Differences Depending on Workload

When choosing the benchmarks, a guiding factor was the amount of load on the bus. This is to evaluate how SystemC coupling reacts to different kinds of workloads. The Ackermann and Sieve benchmarks should do more work on memory than the Sieve

benchmark. Curiously, there is no considerable difference in added overhead when running the benchmarks on a Timing Simple CPU model. Only the O3 and OVPsim models show the expected behavior, here the Monte Carlo benchmark runs with less overhead compared to the others.

Looking at memory access pattern by observing the simulated memory, it became clear that the majority of memory access for all benchmarks is not on data but on instructions. This explains the observed numbers. The Timing Simple model does not use caches, but the O3 models does. While the Timing Simple model needs to access memory for every instruction, the O3 model can take advantage of its cache. Only on the O3 model do different programs observe different memory access patterns.

5.1.5 Consequences

SystemC introduces an overhead to simulation time that cannot be ignored. Especially in the case of OVPsim, where Kras adds up to three orders of magnitude to simulation time. However, the cost might be worth it: Coupling with SystemC allows for way more dynamic and interoperable designs. For example, it is possible to set up complex memory configurations in SystemC and then investigate how different processor models behave without the need to re-implement the memory setups for different simulators. Complex SystemC implementations of peripherals might not be the fastest, but the programmer can fine-tune their models to output just the right level of diagnostic information.

The result of this first step of evaluation has to be that a system designer that couples existing simulators with SystemC will have to expect slowdowns. Depending on the CPU model in use, it can be significant how SystemC models of peripherals are implemented. The bus and memory used for this evaluation were implemented naively, the bus in particular introduced linear search time with every request. Reducing the runtime of peripherals can be helpful in reducing simulation time, though judging from the profiling result, will not make a drastic change as unavoidable SystemC overhead and coupling logic of the simulation cores themselves makes up the majority of added simulation time.

5.2 Goals

The initial discussion in chapter 1 made out three design goals that should be followed in the design of Kras. This section takes a look at them again and rates the implementation on compliance.

5.2.1 Extensibility

Kras is supposed to be extensible, that is it should be easy to extend to different simulators.

This goal motivated SystemC as a glue that connects the different models and components. In the cases of gem5 and OVP, both of which already have a SystemC interface, implementation with Kras was simple and required little extra work. The described approach for integrating GPGPU-Sim with Kras on the other hand requires an additional guest runtime library and a matching proxy SystemC module that forwards requests to the actual GPGPU-Sim implementation. This demonstrates that integration with non-SystemC components is possible but more involved. As such, Kras is at least as extensible as the existing virtual platforms built upon SystemC.

While it is somewhat easy to integrate SystemC modules into Kras, the current approach is not perfect. SystemC modules need to be annotated manually for them to work in conjunction with Kras. This extra step impedes extensibility as it means extra work, extending Kras becomes more laborious than it could be. It would be better if Kras offered facilities for parsing meta data out of SystemC files. External annotations in particular proved to be problematic when using the graphical editor Gras; importing a configuration without importing the matching annotations means that Gras does not know how to represent the interface of the different interfaces.

5.2.2 Abstract Configurations

The second goal of Kras was that it should work with abstract configurations. The following points illustrate how this goal manifests itself in the final implementation and discusses how effective these decisions were in keeping the focus on the abstract design and not on detail:

- Configurations are stored in INI files and condensed to the most essential. This should make it harder to over-complicate the design description with details compared to using full programming languages like SystemC itself to define configurations.

  On the downside, even just representing bus mapping required some special syntax (subsection 3.2.4). Should there ever arise the need to extend Kras configuration files, it might become harder and harder to work with the existing INI format. Maybe the desire for abstract configurations stands in contrast with flexibility for new features in the future.

- Part of the implementation is a graphical interface, Gras. It is easy to configure configurations using Gras and does not require tweaking C++ code. Figure 4.3 is a nice example of how a configuration can be displayed in an easy to grasp way.

While the way configurations are represented is satisfactory, generating a simulation is not as convenient as it should be. This is because, as it stands, Kras configurations are a "*leaky abstraction*" [68], an abstraction above SystemC where SystemC details shine though and need to be taken care of by the user. The problems are the following:

- Constructor parameters required for instantiation have a certain C++ type. Depending on the C++ type, literals need to be formatted differently, e.g. a `std::string` needs to be quoted (`"`) while a regular `int` requires no quoting. Users of Kras have to be aware of this and format mapping values accordingly.
- C++ classes can have polymorphic constructors, i.e. multiple constructors with different parameter lists. Kras currently does not support this C++ feature, it assumes that every SystemC module has exactly one constructor.

5.2.3 Cooperability

The final goal was cooperability. Kras as a framework and tool chain should be easy to integrate into other workflows. This goals has been achieved.

For one, implementation as a library means that Kras can be used to build new tools. It was easy to re-use the library logic in both command line tool and graphical editor. It should be similarly easy to integrate the library into other tools, in theory even into full fledged virtual systems. This is the advantages of the shared library approach.

Looking at the existing command line interface, krasc, it works well in scripts and automated context as it is only a simple text-based compiler for INI configuration files. While the library is easy to integrate into other applications, krasc is easy to integrate into scripted workflows because it is only a small component with one designated task rather than one all-encompassing suite.

Chapter 6

Conclusion

SoCs combine different components in one system. During development, simulations help the designer or researcher to make out distinct properties of the final system ahead of time. For simulation, many tools are available that don't necessarily work together; combining these tools for maximum efficiency can be difficult. This thesis demonstrated an approach for integrating different simulator packages into one easy to use editor.

SystemC TLM provides a good base for combining different models into one configuration. It is already supported by some simulation packages and those that aren't can be converted to use SystemC, as was demonstrated for the GPGPU-Sim GPU simulator. While SystemC provides almost all required primitives for SoC design, it does not include a standardized way for describing bus systems. For this, this thesis suggested the concept of bind ranges. Connecting to a bind range socket means configuring the bus. For simulation, Kras translates abstract configurations to SystemC code. This thesis argued that simulations based on SystemC will always require code generation and compilation as SystemC relies on C++ templates.

One hindrance during development was parsing type information out of SystemC files, the current approach of manual annotation proved to be not satisfactory. It should be replaced with an automatic solution, even if difficult to implement. The biggest problem with SystemC coupling was, as evaluation showed, that integration with SystemC can be quite expensive, which needs to be taken into account when choosing a simulation environment.

In summary, it is certainly possible to build a framework that combines different simulation software and makes them easy to configure. This does introduce an overhead which can be significant. A solution like Kras will not be useful in all situations but if flexibility and interoperability are the main concerns, the approach taken with Kras is a reasonable one.

List of Figures

List of Tables

Bibliography

[1] (2002). ECE1767: Design for Test and Testability, University of Toronto, [Online]. Available: http://www.eecg.toronto.edu/~ece1767/notes/pect9.pdf (visited on 2018-07-13) (cit. on p. 1).

[2] F. Samie, L. Bauer, and J. Henkel, "IoT Technologies for Embedded Computing: A Survey," in *Proceedings of the Eleventh IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, ser. CODES '16, Pittsburgh, Pennsylvania: ACM, 2016, 8:1–8:10. [Online]. Available: http://doi.acm.org/10.1145/2968456.2974004 (cit. on p. 1).

[3] P. Harrod, "Testing reusable IP-a case study," in *Test Conference, 1999. Proceedings. International*, IEEE, 1999, pp. 493–498 (cit. on p. 2).

[4] R. Rajsuman, *System-on-a-chip: Design and Test*. Artech House, Inc., 2000, ch. 1 (cit. on p. 2).

[5] R. Kumar, D. M. Tullsen, N. P. Jouppi, and P. Ranganathan, "Heterogeneous chip multiprocessors," *Computer*, vol. 38, no. 11, pp. 32–38, 2005 (cit. on p. 2).

[6] S. Pasricha *et al.*, "Transaction level modeling of SoC with SystemC 2.0," in *Synopsys User Group Conference (SNUG)*, vol. 3, 2002, p. 3 (cit. on p. 2).

[7] A. Hunt and D. Thomas, *The Pragmatic Programmer*. Addison-Wesley, 2000 (cit. on p. 2).

[8] K. Popovici and A. Jerraya, "Virtual Platforms in System-on-Chip Design," in *Design Automation Conference*, 2010 (cit. on pp. 3, 4).

[9] Synopsis. (2018). Platform Architect Datasheet, [Online]. Available: https://www.synopsys.com/verification/virtual-prototyping/platform-architect.html (visited on 2018-10-03) (cit. on p. 3).

[10] IEEE Standards Association and others, "IEEE Standard for standard SystemC language reference manual," *IEEE Computer Society*, 2012 (cit. on pp. 3, 4, 10, 11, 16, 21, 31).

[11] M. Tovar Rascon and P. Elfborg, "Virtual Cycle-accurate Hardware and Software Co-simulation Platform for Cellular IoT," 2017 (cit. on p. 3).

[12] Mentor Graphics, "Vista Virtual Prototyping Datasheet," 2015. [Online]. Available: `https://www.mentor.com/esl/vista/virtual-prototyping/` (visited on 2018-10-03) (cit. on p. 3).

[13] "IEEE/IEC International Standard - IP-XACT, Standard Structure for Packaging, Integrating, and Reusing IP within Tool Flows," *IEC 62014-4 IEEE Std 1685-2009*, pp. 1–373, 2015-03 (cit. on p. 4).

[14] J. A. Swanson. (). Building an IP-XACT Design and Verification Environment with DesignWare IP, [Online]. Available: `https://www.synopsys.com/designware-ip/technical-bulletin/design-verification-environment.html` (visited on 2018-10-03) (cit. on p. 4).

[15] (2018). SystemC, Accellera Systems Inititative, [Online]. Available: `https://www.accellera.org/downloads/standards/systemc` (visited on 2018-07-19) (cit. on pp. 5, 10, 11).

[16] MathWorks, Inc. (2018). HDL Verifier: Verify VHDL and Verilog using HDL simulators and FPGA-in-the-loop test benches, [Online]. Available: `https://www.mathworks.com/products/hdl-verifier.html` (visited on 2018-10-03) (cit. on p. 5).

[17] Arm Limited (or its affiliates). (2018). Arm SoC Designer, [Online]. Available: `https://developer.arm.com/products/system-design/cycle-models/arm-soc-designer` (visited on 2018-10-03) (cit. on p. 5).

[18] G. Hohpe, B. Woolf, *et al.*, "Enterprise integration patterns," 2003 (cit. on p. 5).

[19] W. D. Kelton, J. S. Smith, and D. T. Sturrock, *Simio & Simulation: Modeling, Analysis, Applications*. Learning Solutions, 2011 (cit. on p. 7).

[20] R. Jindal and K. Jain, "Verification of transaction-level SystemC models using RTL testbenches," in *Formal Methods and Models for Co-Design, 2003. MEMOCODE'03. Proceedings. First ACM and IEEE International Conference on*, IEEE, 2003, pp. 199–203 (cit. on pp. 7, 8).

[21] S. Meftali, J. Vennin, and J.-L. Dekeyser, "A fast SystemC simulation methodology fo Multi-Level IP/SoC design," in *IFIP Intl. Workshop on IP Based SoC Design*, 2003 (cit. on pp. 7, 8).

[22] J. Reichardt and B. Schwarz, *VHDL-Synthese: Entwurf digitaler Schaltungen und Systeme*. Walter de Gruyter, 2012 (cit. on p. 8).

[23] L. Cai and D. Gajski, "Transaction Level Modeling: An Overview," in *Proceedings of the 1st IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, ser. CODES+ISSS '03, Newport Beach, CA, USA: ACM, 2003, pp. 19–24. [Online]. Available: http://doi.acm.org/10.1145/944645.944651 (cit. on pp. 8, 9).

[24] C.-Y. (Huang, Y.-F. Yin, C.-J. Hsu, T. B. Huang, and T.-M. Chang, "SoC HW/SW Verification and Validation," in *Proceedings of the 16th Asia and South Pacific Design Automation Conference*, ser. ASPDAC '11, Yokohama, Japan: IEEE Press, 2011, pp. 297–300. [Online]. Available: http://dl.acm.org/citation.cfm?id=1950815.1950882 (cit. on pp. 8, 9).

[25] F. Kesel and R. Bartholomä, *Entwurf von digitalen Schaltungen und Systemen mit HDLs und FPGAs: Einführung mit VHDL und SystemC*. Oldenbourg Verlag, 2009 (cit. on p. 9).

[26] N. Abdessaied, *Design of a Java Simulator for Fast Prototyping of System-on-chip*. LAP LAMBERT Academic Publishing, 2015 (cit. on p. 9).

[27] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 Simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, 2011-08. [Online]. Available: http://doi.acm.org/10.1145/2024716.2024718 (cit. on p. 12).

[28] A. Roelke and M. R. Stan, "Risc5: Implementing the RISC-V ISA in gem5," in *First Workshop on Computer Architecture Research with RISC-V (CARRV)*, 2017 (cit. on p. 12).

[29] (2018). gem5, Git repository. version a66fe6a8, [Online]. Available: https://gem5.googlesource.com/public/gem5 (visited on 2018-07-16) (cit. on pp. 13, 20).

[30] C. Menard, J. Castrillon, M. Jung, and N. Wehn, "System simulation with gem5 and SystemC: The keystone for full interoperability," in *2017 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, 2017-07, pp. 62–69 (cit. on pp. 12, 34, 35).

[31] F. Rosa, L. Ost, R. Reis, and G. Sassatelli, "Instruction-driven timing CPU model for efficient embedded software development using OVP," in *2013 IEEE 20th International Conference on Electronics, Circuits, and Systems (ICECS)*, 2013-12, pp. 855–858 (cit. on p. 13).

[32] (2018). Technology OVPsim, Imperas Software, [Online]. Available: http://www.ovpworld.org/technology_ovpsim (visited on 2018-07-20) (cit. on pp. 13, 35).

[33] (2018). General Information about OVP Models, Imperas Software, [Online]. Available: `http://www.ovpworld.org/technology_models` (visited on 2018-07-20) (cit. on p. 13).

[34] (2018). OVP Technology works well with Accellera (OSCI) SystemC, Imperas Software, [Online]. Available: `http://www.ovpworld.org/technology_systemc` (visited on 2018-07-20) (cit. on p. 13).

[35] K. T. Cheng and Y. C. Wang, "Using mobile GPU for general-purpose computing; a case study of face recognition on smartphones," in *Proceedings of 2011 International Symposium on VLSI Design, Automation and Test*, 2011-04, pp. 1–4 (cit. on p. 14).

[36] A. Bakhoda, G. L. Yuan, W. W. Fung, H. Wong, and T. M. Aamodt, "Analyzing CUDA workloads using a detailed GPU simulator," in *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*, IEEE, 2009, pp. 163–174 (cit. on p. 14).

[37] (2018). gpgpu-sim, Git repository. version `ba33904b`, [Online]. Available: `https://github.com/gpgpu-sim/gpgpu-sim_distribution` (visited on 2018-07-17) (cit. on p. 14).

[38] R. Thomas, "A Parser of the C++ Programming Language," 2005 (cit. on p. 17).

[39] A. Hein, "Identification and bridging of semantic gaps in the context of multi-domain engineering," in *Forum on Philosophy, Engineering & Technology*, 2010 (cit. on p. 17).

[40] A. Van Deursen, P. Klint, and J. Visser, "Domain-specific languages: An annotated bibliography," *ACM Sigplan Notices*, vol. 35, no. 6, pp. 26–36, 2000 (cit. on p. 19).

[41] The Rust Team, *The Rust Programming Language*. 2008-10 (cit. on p. 19).

[42] *Swift*. Apple Inc. (cit. on p. 19).

[43] *Microsoft: Windows NT Workstation Resource Kit*. Microsoft Press, 1996 (cit. on p. 20).

[44] R. Petersen, *Fedora Linux Servers with systemd*. Surfing Turtle Press, 2018 (cit. on p. 20).

[45] (2018). configparser — Configuration file parser, Python Software Foundation, [Online]. Available: `https://docs.python.org/3/library/configparser.html` (visited on 2018-08-12) (cit. on p. 20).

[46] (2011). Java API for handling Windows ini file format, [Online]. Available: `http://ini4j.sourceforge.net/index.html` (visited on 2018-08-12) (cit. on p. 20).

[47] M. Kalicinski and R. Sebastian. (2013). Boost.PropertyTree, [Online]. Available: https://www.boost.org/doc/libs/1_65_1/doc/html/property_tree.html (visited on 2018-07-31) (cit. on pp. 20, 30).

[48] D. Bartholomew, "QEMU: a Multihost, Multitarget Emulator," *Linux Journal*, vol. 2006, no. 145, p. 3, 2006 (cit. on p. 21).

[49] V. David and N. M. Josuttis, "C++ Templates: The Complete Guide," *Addison-Wesley Professional*, 2002 (cit. on p. 22).

[50] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley (cit. on p. 24).

[51] G. Phipps, "Comparing observed bug and productivity rates for Java and C++," *Software: Practice and Experience*, vol. 29, no. 4, pp. 345–358, 1999 (cit. on p. 25).

[52] C. Lattner, "LLVM and Clang: Next generation compiler technology," in *The BSD conference*, 2008 (cit. on p. 28).

[53] (2018). Clang 7 Documentation: LibTooling, [Online]. Available: https://clang.llvm.org/docs/LibTooling.html (visited on 2018-07-31) (cit. on p. 28).

[54] A. Mukherjee, *Learning Boost C++ libraries : solve practical programming problems using powerful, portable, and expressive libraries from Boost*. Packt Publishing, 2015 (cit. on p. 30).

[55] D. Große, R. Drechsler, L. Linhard, and G. Angst, "Efficient Automatic Visualization of SystemC Designs.," in *FDL*, 2003, pp. 646–658 (cit. on p. 32).

[56] R. Drechsler, G. Fey, C. Genz, and D. Große, "SyCE: An integrated environment for system design in SystemC," in *Rapid System Prototyping, 2005.(RSP 2005). The 16th IEEE International Workshop on*, IEEE, 2005, pp. 258–260 (cit. on p. 32).

[57] B. Jasmin and M. Summerfield, *C++ GUI Programming with Qt4*. Prentice Hall, 2008 (cit. on p. 32).

[58] R. Lischner, *STL: pocket reference*. O'Reilly, 2009 (cit. on p. 33).

[59] (2018). Model/View Programming, The Qt Company Ltd., [Online]. Available: https://doc.qt.io/qt-5/model-view-programming.html (visited on 2018-08-01) (cit. on p. 33).

[60] R. Jain. (2011). Types of Workloads, [Online]. Available: https://www.cse.wustl.edu/~jain/cse567-11/ftp/k_04tow.pdf (visited on 2018-08-21) (cit. on p. 40).

[61] Y. Sundblad, “The Ackermann Function. A Theoretical, Computational, and Formula Manipulative Study,” *BIT Numerical Mathematics*, vol. 11, no. 1, pp. 107–119, 1971 (cit. on p. 40).

[62] J. Sorenson, “An Introduction to Prime Number Sieves,” ComputerScience Technical Report, Tech. Rep., 1990 (cit. on p. 40).

[63] K. Achilles, “Monte-Carlo-Pi,” in *BASIC und Pascal im Vergleich*, Springer, 1983, pp. 72–74 (cit. on p. 40).

[64] (2018). Coremark: An EEMBC Benchmark, [Online]. Available: `https://www.eembc.org/coremark/` (visited on 2018-09-29) (cit. on p. 40).

[65] (2017). Callgrind: a call-graph generating cache and branch prediction profiler, [Online]. Available: `http://valgrind.org/docs/manual/cl-manual.html` (visited on 2018-09-29) (cit. on p. 42).

[66] S. L. Graham, P. B. Kessler, and M. K. Mckusick, “Gprof: A Call Graph Execution Profiler,” in *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction*, ser. SIGPLAN ’82, Boston, Massachusetts, USA: ACM, 1982, pp. 120–126. [Online]. Available: `http://doi.acm.org/10.1145/800230.806987` (cit. on p. 43).

[67] S. Wolfman. (1999-09). Profiling with GProf, [Online]. Available: `https://users.cs.duke.edu/~ola/courses/programming/gprof.html` (visited on 2018-09-29) (cit. on p. 43).

[68] J. Spolsky, “The Law of Leaky Abstractions,” in *Joel on Software*, Springer, 2004, pp. 197–202 (cit. on p. 46).