

A LOOSELY-COUPLED GRAPHICAL USER INTERFACE FOR RUN-TIME CONTROL OF SYSTEMC SIMULATION MODELS

CARSTEN ALBRECHT, CHRISTIAN J. EIBL*, RAINER HAGENAU†

Institute of Computer Engineering

University of Lübeck

Ratzeburger Allee 160, D-23538 Lübeck, Germany

Email: {albrecht, eibl, hagenau}@iti.uni-luebeck.de

Abstract: Modern hardware development bases on high-level methods with appropriate tool support. SystemC, a C++ class library, provides a high-level interface to model and simulate hardware designs on different levels. Unfortunately, there is no graphical interface included for demonstration, debugging, or educational purposes. *gSysC* presented here forms a GUI for SystemC. It allows the programmer to watch the interaction of the simulated design parts and provides additional run-time control features such as single-step simulation or breakpoints. The application of *gSysC* to SystemC models is demonstrated using some examples. Further on, the overhead of *gSysC* applied to a selection of the example designs is measured and compared to the model complexity.

Keywords: SystemC, GUI, Simulation Controller

1 INTRODUCTION

Simulation is a state-of-the-art process to test, verify, and profile newly designed hardware models. Generally, it provides exhaustive views and in-depth analysis of crucial, unapproachable system parts. Nowadays, an increasing number of hardware design tools using hardware description languages are available. But there is a demand for higher-level methods of modelling and simulating, especially for hybrid hardware/software systems. SystemC [?] brought out by a pool of companies is a C++ class library that allows simulation of systems compounded of modules mod-

elled on varying abstraction levels. It backs the top-down design methodology so that each module can be iteratively redesigned. Unfortunately, SystemC models can only be analysed by trace and log files. Visualisations of simulated modules and their interaction is not provided but they can be of course introduced utilising any library for graphical output. A fixed graphical user interface (GUI) would be helpful to benefit even more of SystemC simulation models. Especially in the areas of presentation, demonstration, and education, visual support is absolutely useful.

In the following, related systems are introduced. Their strengths and weaknesses are shortly discussed and, in contrast to them, the benefits of *gSysC* are explained. Moreover, the main details are shown presenting first the concept, further on, its implementation and way of application. Different case studies show the application of *gSysC* and its features. Finally,

*now with the Research Group of Didactics and eLearning at the Department of Electrical Engineering and Computer Science, University of Siegen, Hölderlinstraße 3, D-57076 Siegen, Germany, eibl@die.informatik.uni-siegen.de

†now with hagenau system competence, Willy-Brandt-Allee 31b, 23554 Lübeck, Germany, info@hagenau-sc.de

the performance loss of applying *gSysC* to a selection of these examples is examined.

2 RELATED WORK

As mentioned above, SystemC is a C++ class library and is applied for system modelling and simulation on various levels. Due to its software architecture add-ons can be easily attached to or combined with libraries so that the functionality of SystemC designs are extended by proper requirements. The SystemC core remains untouched. An example is the OSSS+R [?] library which extends the SystemC core language by statements for partially run-time reconfigurable devices such as current field programmable gate arrays, e.g. the Xilinx Virtex4 family [?]. In addition to the simulation features of SystemC, it bases on a synthesisable subset of SystemC and provides hardware mechanisms to support run-time reconfigurability. With regard to the properties of C++, it forms a fully object-oriented hardware design environment.

As seen before, the design flow with SystemC may even include the synthesis of the design. Thus, it is obvious that tool support for code generation, debugging, analysis, and profiling is necessary. Recently, appropriate integrated development environments (IDE) appeared. A centric part of those tool suites is the GUI that visualises the SystemC model and provides design and simulation control.

There are both commercial and non-commercial approaches to a GUI for SystemC. Commercial approaches generally integrate SystemC into their hardware design environments. It is used for fast functional analysis and verification. For example, Prosilog Magillem [?] allows the user to automatically generate SystemC as well as Verilog and VHDL code for the graphical hardware design. Moreover, simulators such as the Incisive Unified Simulator from Cadence [?] support SystemC as well. They are able to simulate mixed-language designs so that test benches can be easily defined using SystemC. But the simulation engine is a proprietary one.

A GUI based on the open SystemC library is described in [??]. It is a self-made, Qt-based front end. Qt is a platform independent GUI

library for C++ [?]. The focus is laid on the graphical run-time observation of signals. The interconnection of GUI and SystemC is implemented by adapting the SystemC library so that the GUI is notified of signal-value alterations by the simulation engine. The GUI provides a simulation controller for configuration, initialisation, and run-time control. In contrast to this system, the GUI presented in [?] shows the full system architecture using the interactive visualisation tool SpiceVisionTM. All information on system model and run-time signal values is extracted by a modification of the SystemC library. In combination with other self-made tools for e.g. debugging or verification it forms the IDE SyCE [?].

So, both visualisation systems have a firm bond to the SystemC library because of the necessary modifications. Since 1999, several updates and revisions of SystemC were released so that both GUIs require high effort to keep them up to date.

3 CONCEPT OF *gSysC*

In contrast to these tightly-coupled approaches, *gSysC* provides a loosely-coupled GUI for SystemC. It is based on Qt as well and can be fully removed by a compiler flag without changing the code.

The main goals of *gSysC* are independence of different releases and of the provision of the programmer's permission, free selection which parts are displayed, and an option of removing the visualisation. Additionally, because of the high portability of SystemC, the GUI should be supported by most platforms supporting SystemC.

Figure ?? shows the interfaces between user, SystemC model, SystemC simulation kernel represented by white boxes and the *gSysC* extensions in grey boxes. The user interface of SystemC is limited to reading a configuration at the simulation start and writing textual information or signal traces to the console or hard disk. The user cannot interact with the running simulation. The SystemC design is the software model of the simulated hardware and SystemC represents the simulation kernel and library. *gSysC* based on the graphic library provides besides graphical presentations of the design run-time access to the

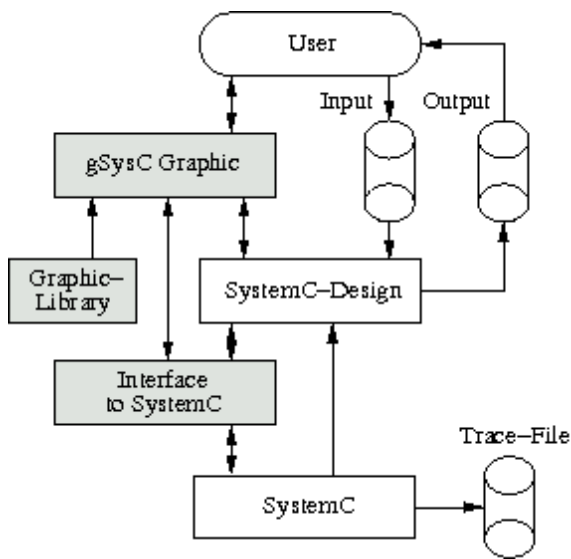


Figure 1: Programmer's View of SystemC and *gSysC*.

simulation. It introduces a simulation controller shown in Figure ?? that provides single-step simulation, simulation of a certain number of clock cycles, or conditional break points, e. g. the simulation halts at a certain signal value. For an automatic and continuous simulation the time delay of each cycle can be determined. Further on, the GUI shows the simulated models with its modules and allows the user to browse the different levels and properties of modules and signals.

The GUI is attached to the simulation program by preprocessor macros. They register each module and signal ports as well as their relation to others. Especially for demonstration purposes, the programmer can focus on important parts and leave out less important ones. The macros include code to indicate any value changes to the GUI kernel. The intervals of indication can be configured as a number of clock cycles.

gSysC takes control of the simulation system by overloading SystemC's control functions such as `sc_start()`. A base set of library methods can be used to show certain details within a simulation run, e.g. fill level of buffers. Due to the variety of simulation opportunities the programmer can enlarge this set by own application-specific extensions.

In case of an almost bug-free SystemC simu-

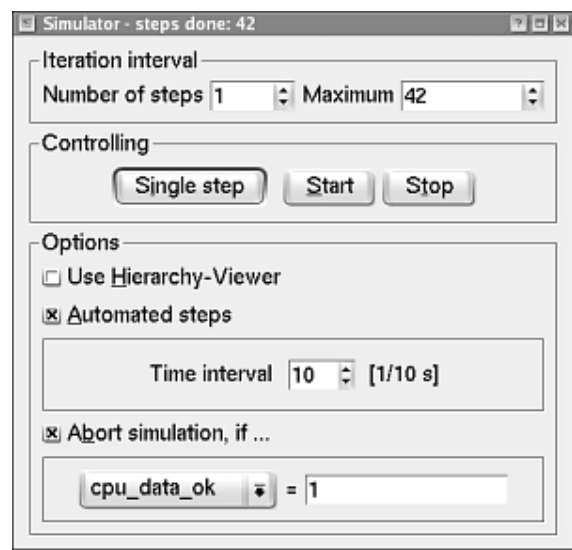


Figure 2: Simulation Controller.

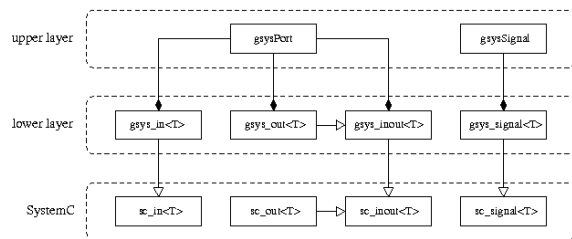


Figure 3: Architecture of the Interface Between *gSysC* and SystemC.

lation model, *gSysC* can be removed at compile time by setting a define flag. Then, an unmodified SystemC simulation is created. This is suitable for long-term runs in particular.

4 IMPLEMENTATION AND APPLICATION

The implementation of *gSysC* has to perform a balancing act between SystemC, Qt and its goals previously defined. The avoidance of additional programmer's code for visualisation leads to a hierarchical structure for the interface between SystemC and *gSysC*. Nevertheless, applying *gSysC* to SystemC models requires some additional code lines.

4.1 Interface Between *gSysC* And SystemC

The interface architecture is shown in Figure ?? including all classes and their relations. It is divided into two layers. In the lower layer, which is next to SystemC, port and signal classes of SystemC are derived in order to receive new values written to them. In the upper layer there are port and signal equivalents of *gSysC* for processing purposes in *gSysC*. In addition to the values, the *gSysC* classes provide information and functions for visualisation. The included functions perform port and signal highlighting, emphasize the position and the connected neighbour modules, and open property information windows. The lower layer contains all classes derived from SystemC. These wrapper classes are necessary to achieve a data-type independent implementation and to get the opportunity to easily use sets and lists, even with ports and signals of different data types. Access to the SystemC layer is only performed by the derived *gSysC* classes so that all value changes can be tracked.

4.2 Registration

Hardware modules may have a huge number of interfaces for control and data exchange purposes. So, there are a couple of reasons to leave out some ports and signals respectively in the visualisation. Fewer observed objects ease the overview of the demonstrated design, allow easier debugging by concentrating on the chosen view, and speed up the simulation. All tracked parts of the design must be registered for the visualisation. So, the registration by hand can become a clumsy procedure, for large designs in particular. The programmer has to add a code line per module or port that activates *gSysC* features for this object and integrates it into the hierarchical structure of the design. The provided macros reduce the effort to a minimum:

- `REG_MODULE(module, name, parent)` registers the `module` on the subsequent `parent` level. Root-level modules are indicated by the `NULL` pointer. The `name` is used in the visualisation.
- `REG_PORT(port, module, signal)` activates the visualisation of the used

ports in the SystemC design. The `port` of `module` is connected to the `signal`. More distinguished macros such as `REG_IN_PORT(port, module, signal)`, `REG_OUT_PORT(port, module, signal)`, and `REG_INOUT_PORT(port, module, signal)` including the direction of the ports are available as well.

- `RENAME_SIGNAL(object, name)` and `RENAME_PORT(object, name)` allow the programmer to give signals and ports self-documenting names.

A further reduction of registration effort is a *gSysC* preprocessing tool. This preprocessor scans the code twice. First, it looks for SystemC structures such as modules, ports, and signals attributed with their relation to each other. Second, the registration for every SystemC structure is inserted. Additionally, the SystemC library is replaced by the *gSysC* library. So, after the preprocessing step, a usual SystemC design can be completely demonstrated and controlled by the GUI features of *gSysC*. Unfortunately, optional and application-specific extensions of *gSysC* have to be included manually. The removal of certain system parts from the visualisation can now be performed by deactivating the registration code using e.g. comments or C preprocessor macros.

5 EXAMPLES

In the following, *gSysC* is demonstrated using three different designs. The first one is a simple producer-consumer model that is used in SystemC tutorials as well. It nicely shows the similarity of block-level system diagrams and *gSysC* representations at a low level. The second one presents the memory hierarchy of general-purpose processor. The cache is tightly coupled to the processor and connected to a memory module by a bus. The hand-made registration is demonstrated in detail. Finally, a model of a network-processor design is shown applying a single object several times.

5.1 Producer-Consumer Model

The producer-consumer model demonstrates the functionality of a first-in, first-out (FIFO) queue.

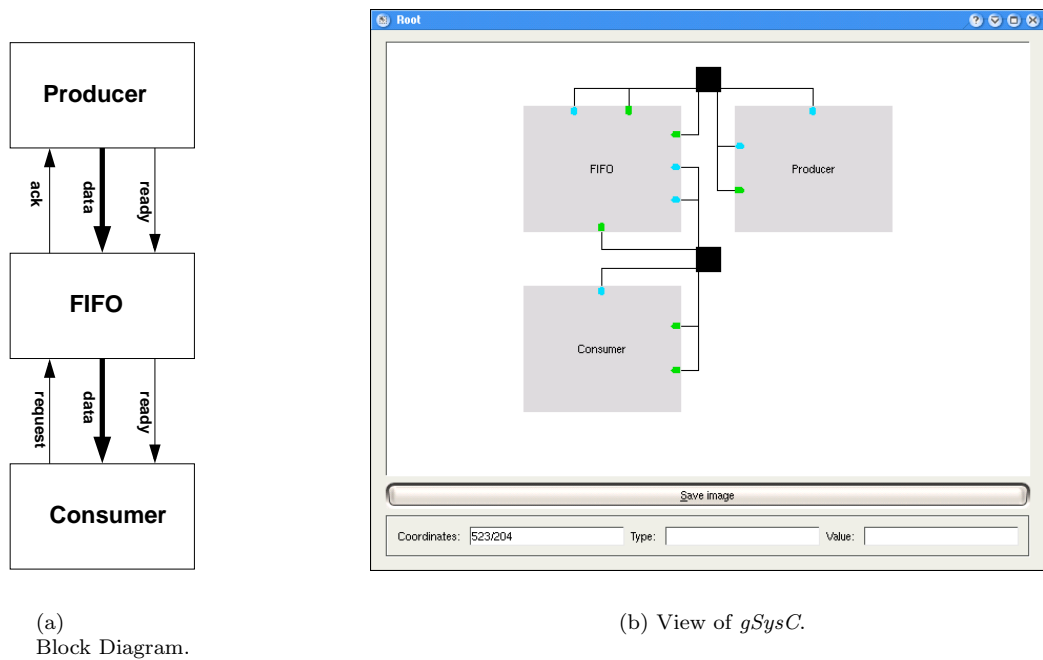


Figure 4: Producer-Consumer Model Views.

The FIFO is utilised to interconnect two components with diametrical interfaces. Both producer and consumer trigger their interfaces actively so that a passive buffer is needed to adapt them. The system shown by a block diagram in Figure ?? acts as follows: the producer randomly generates data that is immediately sent to the consumer. The consumer requests data at randomly-chosen points of time. The necessary synchronisation is performed by the FIFO that takes and delivers the generated data on demand.

The communication bases on simple shake-hand protocols. The producer puts the generated data on its **data** line, sets the **ready** signal, and waits for the FIFO. If the **ack** signal is set by the FIFO, the **ready** signal is reset and the producer goes on. The consumer asynchronously requests data by setting the **request** signal. The FIFO answers by putting data on the **data** signal and sets the **ready** signal to announce a new value. If the consumer resets the **request** data and **ready** signals are freed as well. Note that the FIFO has a limited capacity so that it may happen that requests on any side of the interface are

	Port name	Value	ID
1	F_cdata	575705360	135710520
2	F_pdata	1967632854	135710144
5	F_ready	1	135711816
6	F_data_ok	0	135712232
7	P_data	1967632854	135698840
8	P_data_ok	0	135699648
10	C_data	575705360	135723608
11	C_ready	1	135725984

Figure 5: Table of Ports for Value Tracking.

not instantly served and the requesting instance is stalled.

The equivalence of Figure ?? and Figure ?? is obvious. Both representations consist of three components and an equal number of signals and ports. In Figure ??, the port directions are shown

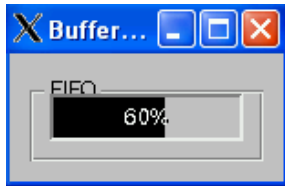


Figure 6: Buffer Level.

by different colours. Names and types of ports and signals are given by setting the mouse focus on it.

Further on, one can keep an eye on port values utilizing their property windows or watching a selection of ports. A snapshot of this utility is shown in Figure ???. The left column shows the port names, in the middle their current values are shown, and in the right column a unique identifier is given. It is used to distinguish multiple but identical names. Additionally, the identifier allows a mapping to the port information of the hierarchy browser.

Another feature of *gSysC* are application-specific extensions. Here, a fill-level view of a buffer is introduced. It provides useful information of a complete component at a glance. Figure ??? shows the fill level of the FIFO during the simulation run. Nevertheless, more complex components cannot be described by a single value so in general the development of extensions similar to this one is left to the programmer.

5.2 Cache-Bus-Memory Model

The hardware design of this SystemC model consists of a CPU directly connected to a memory cache and bus modules. The cache sends to and receives data from the RAM module using a simple bus. Figure ??? shows all modules of the top level model. Here, one can see how the modules with ports and interconnections are displayed. The place and route strategy is simplified using clustered signals with central crossing points. During the simulation used signals, ports, and modules may be highlighted. Figure ??? discloses the module 'Cache-Connect' of Figure ???. It is made of two linked modules unseen on the top level view and has a number of in and out ports shown on the left side. If the view of a module

is opened, its interior behaviour is highlighted as well during the simulation.

The simplified program code of the SystemC model presented here shows the important parts of applying *gSysC*:

```
#include "gsysc.h"
#include ...

int sc_main(int argc, char* argv[])
{
    sc_clock cpu_clk("CPU-Clock");

    // signal declarations
    sc_signal<sc_bv<32> > addr_sig;
    sc_signal<bool> we_sig;
    ...
    cache_connect* c;
    ...

    REG_MODULE(c, "Cache-Connect", NULL);
    REG_MODULE(c->ctrl, "CController", c);
    REG_MODULE(c->memory, "CMemory", c);

    sc_signal<bool> bus_clk_sig;

    bus_bus b("bus");
    b.m_dt(or_mb_dt);
    b.clk(bus_clk_sig);

    REG_MODULE(&b, "Bus", NULL);
    REG_INOUT_PORT(&b.m_dt, &b, &or_mb_dt);
    ...

    c->cpu_clk(cpu_clk);
    c->bus_clk(bus_clk_sig);

    REG_IN_PORT(&c->cpu_clk, c, &cpu_clk);
    REG_IN_PORT(&c->bus_clk, c, &bus_clk_sig);

    bus_master_or mor("master_or");
    REG_MODULE(&mor, "Bus-Master-OR", NULL);
    ...

    return 0;
}
```

First, clock and signals for module interconnection are declared. Then, the modules are defined and their ports are connected to the signals. Last, the module and its ports are registered.

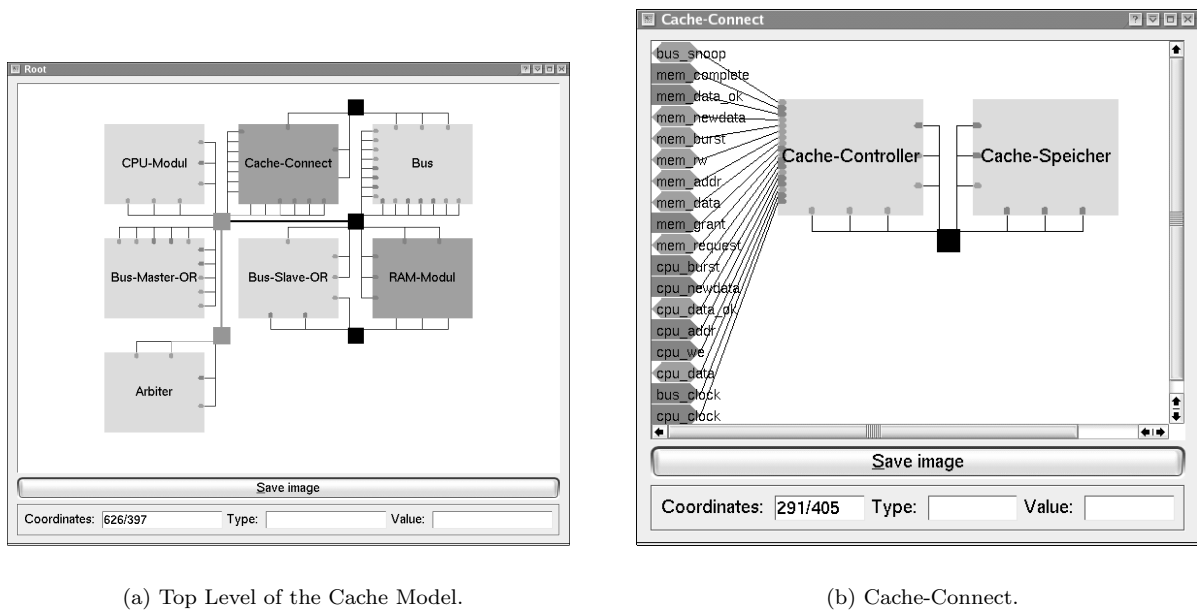


Figure 7: Visualisation of Two Hierarchy Levels in the SystemC Cache Model.

Here, `cache_connect` is built of a controller and memory so that these modules are registered for a subsequent level. These three steps are done for all modules. After the declaration, connection, and registering phases, the usual SystemC code which is not shown here is required.

5.3 Network Processor

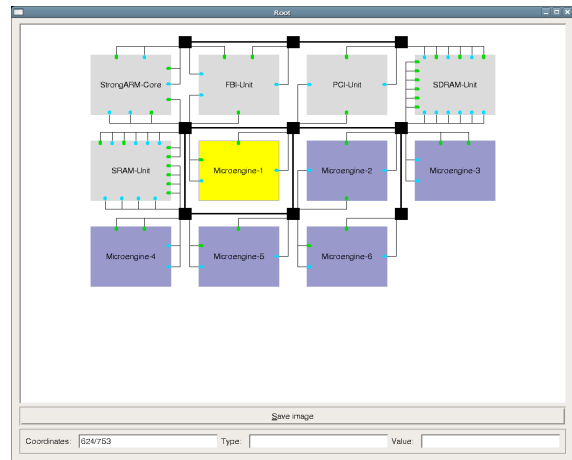
A network processor is an on-chip multiprocessor system including several features for network processing. It generally consists of the control plane, the data plane, and an on-chip communication infrastructure. The control plane performs less time-critical tasks and is basically realised by a general-purpose processor. The data plane has to process incoming data at wire-speed. Since software-programmable devices do not provide enough performance to keep up with current link speed, on-chip multiprocessor systems built up by application-specific processors are used. Buses and direct connections form the on-chip communication infrastructure. Here, a simplified model of the Intel IXP1200 [?] is introduced. Figure ?? shows some *gSysC* views of the model. Figure ?? lists all included modules displaying

the instances shown by the other subfigures of Figure ?? . Figure?? displays the root level of the IXP model. A StrongARM core is used for the control plane, the Microengines 1 – 6 set up the data plane. The FBI and PCI unit are bus interfaces for internal and external communication. The SRAM and the SDRAM unit provide memory access to small and fast or big but slow memory modules.

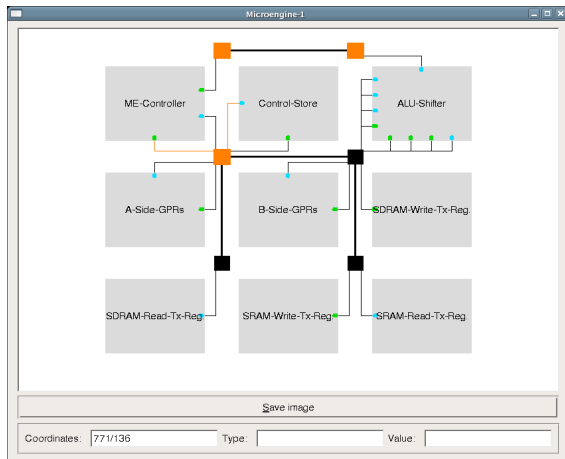
Here, the more interesting parts are the microengines on a lower hierarchy level. They consist of a controller, a control store, an ALU shifter, two sets of general-purpose registers and several transfer registers for read/write access to the memories. All microengines execute code of different threads. Thus, a different behaviour of the microengines is expected for the same point of time. Figure ?? and ?? show the concurrency of these special-purpose processors. The highlighted parts are active. All in all, *gSysC* allows the user to get a quick overview of the complex design and to follow multiple objects of the same type at a glance. Debugging as well as comprehension of the design can be eased by the visualisation.

Module	Type	Address
Root	Root	0
-FBI-Unit	Modul	136192888
-Microengine-1	Modul	136193432
-A-Side-GPRs	Modul	136193976
-ALU-Shifter	Modul	136193840
-B-Side-GPRs	Modul	136194112
-Control-Store	Modul	136193704
-ME-Controller	Modul	136193568
-SDRAM-Read-Tx-Reg.	Modul	135583544
-SDRAM-Write-Tx-Reg.	Modul	136194248
-SRAM-Read-Tx-Reg.	Modul	135583944
-SRAM-Write-Tx-Reg.	Modul	135583744
-Microengine-2	Modul	136195560
-Microengine-3	Modul	136197176
-A-Side-GPRs	Modul	136197720
-ALU-Shifter	Modul	136197584
-B-Side-GPRs	Modul	136197856
-Control-Store	Modul	136197448
-ME-Controller	Modul	136197312
-SDRAM-Read-Tx-Reg.	Modul	136198192
-SDRAM-Write-Tx-Reg.	Modul	136197992
-SRAM-Read-Tx-Reg.	Modul	136198856
-SRAM-Write-Tx-Reg.	Modul	136198656
-Microengine-4	Modul	136199056
-Microengine-5	Modul	136200672
-Microengine-6	Modul	136202288
-PCI-Unit	Modul	136193024
-SDRAM-Unit	Modul	136193160
-SRAM-Unit	Modul	136193296
-StrongARM-Core	Modul	136192752

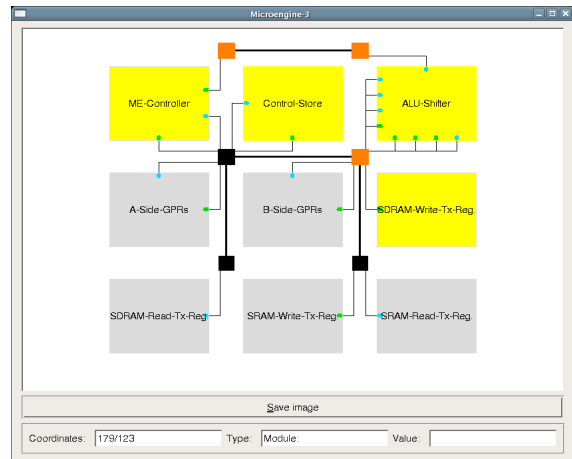
(a) Module Overview.



(b) Root Level.



(c) Microengine 1.



(d) Microengine 3.

Figure 8: Visualisation of IXP1200 Model.

6 PERFORMANCE STUDIES

It is obvious that an extension requires additional resources. Thus, the program uses more space on disk and memory and its execution time slows down. The performance-loss for applying *gSysC* is approximately determined using the SystemC models of the FIFO and the cache-bus-memory model. Their complexity strongly differs. The FIFO consists of a flat hierarchy with three components connected by six signals. Whereas the cache model includes two hierarchy levels, comprising of seven components on the first and two components on the second level respectively, and 34 signals for module interconnection.

The measurements are performed on a dual Pentium III, 1 GHz with 256 kB cache and 512 MB memory. Both programs are compiled in two different ways: a native SystemC program using the compiler option of *gSysC* and a *gSysC* version. Runs with *gSysC* only use the controller functionality, signal highlighting, application-specific extensions were not included. The automatic stepping introduces breaks to depict the changes. The length of the break is defined by the user and has the major impact on the run-time. Therefore, the measurement is limited to user-independent simulations.

Figure ?? shows the absolute run-time of all resulting programs simulating up to 250000 clock cycles. The empty boxes and diamonds show the native SystemC simulations and the filled ones show the *gSysC* variants. In both cases, the run-time of the *gSysC* variants are definitely slower. The start-up phase up to 7500 clock cycles is basically the same. Here, the duration of execution does not differ at all. With regard to the simulated designs, both systems are filled with randomly generated data. Major changes of the internal behaviour do not occur. Beyond the start-up phase, the run-time increases approximately linearly. The gap between the native SystemC and *gSysC* variant of single model is compared in Figure ?. The shown run-time ratio (*RTR*) is computed by

$$RTR = \frac{runtime_{gSysC}}{1 + runtime_{SystemC}}.$$

The gap of the FIFO model is rather big com-

pared to the cache model although the number of tracked FIFO entities is smaller. So, the overhead of *gSysC* is quite huge for less complex models, complex models with longer SystemC execution times levels the overhead out.

7 CONCLUSION

gSysC is a GUI extension for SystemC based on Qt, a platform independent GUI library for C++. The shown extension does not alter the SystemC library including the simulation kernel. The features for graphical representation are introduced by macros and redefined functions overloading but calling the ones provided by SystemC.

In combination with the existing opportunities such as VCD trace files and text messages, SystemC becomes with *gSysC* a more powerful tool for debugging and functional verification. Nevertheless, the costs of *gSysC* are partially high. The simulation run-time slows down depending on the model complexity and the user configuration. Because of the opportunity to remove *gSysC* from the SystemC-model code at compile time, long-term simulations without GUI are executed without any performance loss.

The library is open source and can be found at the web pages of the Institute of Computer Engineering, University of Lübeck (www.iti.uni-luebeck.de).

REFERENCES

- Cadence Design Systems Inc. 2005. *Incisive Unified Simulator*, Datasheet 5418C 04/05.
- Charest L., Reid M., Aboulhamid E.M., and Bois G. 2001. *Methodology for Interfacing Open Source SystemC with a Third Party Software*. Proceedings of Design Automation and Test in Europe Conference & Exhibition, Munich, Germany. Pp16-20.
- Drechsler R., Fey G., Genz C., and Große D. 2005. *SyCE: An Integrated Environment for System Design in SystemC*. 16th IEEE International Workshop on Rapid System Prototyping (RSP), Montreal, Canada.

Eibl C.J. 2004. *gSysC – Visualisierung von SystemC-Projekten*. Student Project, Institute of Computer Engineering, University of Lübeck, Germany.

Große D., Drechsler R., Linhard L., and Angst G. 2003. *Efficient Automatic Visualization of SystemC Designs*. *Forum on Specification & Design Languages*, Frankfurt, Germany.

Intel Corporation 2001. *Intel IXP1200 Network Processor*. Datasheet, Part Number 278298-010.

Prosilog 2005. *Magillem*. Product Brief.

Reid M., Charest L., Aboulhamid E.M., Bois G., and Tsikhanovich A. 2001. *Implementing a Graphical User Interface for SystemC*. Proceedings of the 10th International HDL Conference, Santa Clara, CA, USA. Pp224-231.

Schallenberg A., Oppenheimer F., and Nebel W. 2004. *Designing for dynamic partially reconfigurable FPGAs with SystemC and OSSS*. In Proceedings of FDL'04. University of Oldenburg, Germany.

Open SystemC Initiative (OSCI) 2002. *SystemC Version 2.0.1 User's Guide*. Technical Report.

Trolltech AS 2002. *Qt 3.1*. Whitepaper.

Xilinx Inc. 2005. *Virtex-4 Family Overview*. Datasheet.

AUTHOR BIOGRAPHIES



Carsten Albrecht received his Diploma degree in Computer Science from the

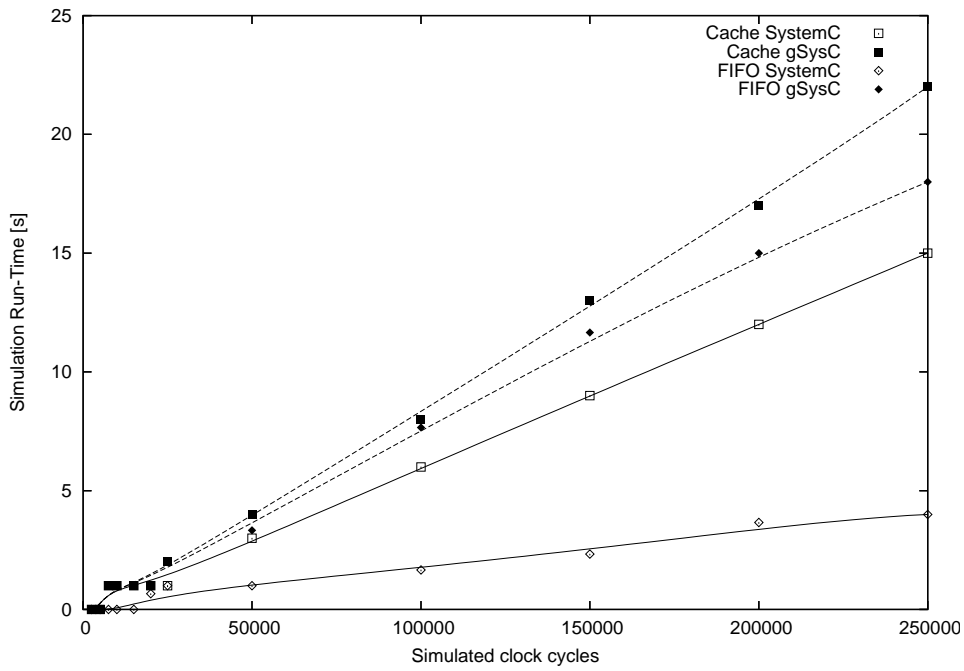
University of Lübeck in 2002. In the same year, he joined the Institute of Computer Engineering, University of Lübeck, as a research associate. His current research topic is application-specific management of dynamically reconfigurable systems. Further research interests include multithreading and network processor architectures.



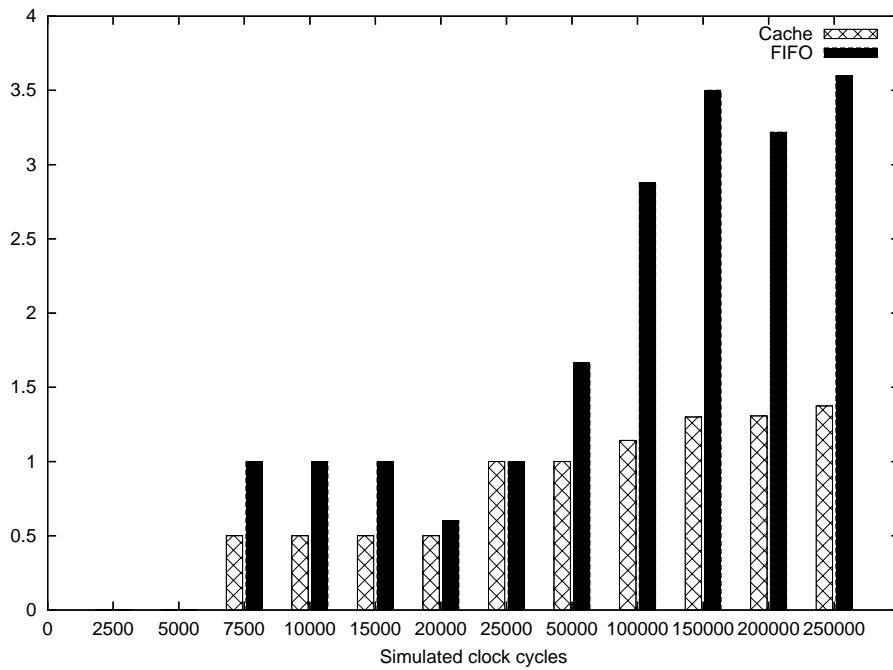
Christian Josef Eibl received his Diploma degree in Computer Science from the University of Lübeck in 2005. In the same year, he joined the research group of Didactics in Computer Science and eLearning at the University of Siegen as a research associate. His current research focus is on eLearning theory and motivation of learners. Further research interests include cryptography, security engineering, and visualisation for easier comprehension of complex structures.



Rainer Hagenau received his Diploma degree in Computer Science from the University of Berlin in 1991. After employments in a manufacturing company for medical devices and in the retail industry, he joined the Institute of Computer Engineering, University of Lübeck, as a research associate in 1998. Here, his research topics were parallel computing and network processor architectures. In 2005, he founded hagenau system competence.



(a) Absolute Run-Time of the Cache and FIFO Models with Native SystemC and *gSysC*.



(b) Run-Time Ratio (*RTR*) of Cache and FIFO Model Comparing Native SystemC with *gSysC*.

Figure 9: Comparison of SystemC and *gSysC* Simulation Run-Time.