CARL VON OSSIETZKY universität OLDENBURG

Fakultät II – Informatik, Wirtschafts- und Rechtswissenschaften
Department für Informatik

Dynamic Partial Self-Reconfiguration: Quick Modeling, Simulation, and Synthesis

Von der Carl von Ossietzky Universität Oldenburg
- Fakultät II (Department Wirtschafts- und Rechtswissenschaften) -
zur Erlangung des Grades eines

Doktors der Ingenieurwissenschaften (Dr.-Ing.)

genehmigte Dissertation

von Herrn

Dipl.-Inform. Andreas Schallenberg

geboren am 21. August 1974 in Osnabrück.

Tag der Disputation: 12. Mai 2010

Erstgutachter:
Prof. Dr.-Ing. Wolfgang Nebel
Carl von Ossietzky University Oldenburg

Zweitgutachter:
Prof. Dr. Marco Platzner
University of Paderborn

Drittgutachter:
Prof. Dr. rer. nat. Achim Rettberg
Carl von Ossietzky University Oldenburg

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Some decades ago a circuit designer had little choices when implementing a new system. If flexibility was needed, it was built around general purpose processors (GPPs). Demand for more speed meant building custom solutions, so application specific integrated circuits (ASICs) were chosen. Over the years, more and more alternatives appeared on the market. Digital signal processors (DSPs) were designed for data-flow intensive computation. Glue logic grew to programmable logic arrays (PLAs) and complex programmable logic devices (CPLDs).

Along with them, field programmable gate arrays (FPGAs) were introduced. Like all the other techniques, FPGAs fill a gap. From a designers point of view, they can be seen as programmable ASICs offering less speed and less power efficiency. Still, they offer the same concurrency in calculation as ASICs do. Figure 1.1 sketches, how some popular device types relate to each other.

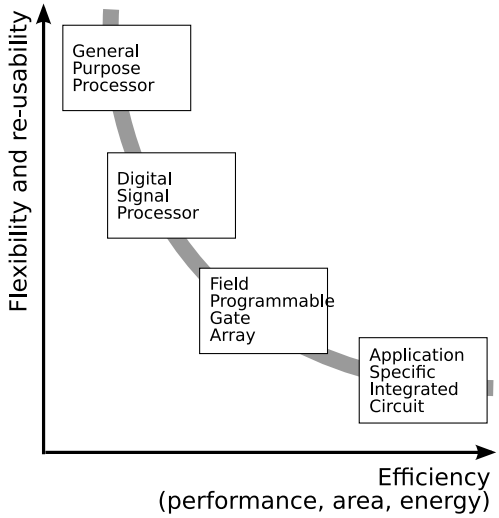

Figure 1.1: Trade off between flexibility and efficiency

FPGAs first became popular for rapid prototyping. When used inside low-volume products, they are now more cost-efficient than ASICs. As FPGA prices are continually dropping and with ASIC mask costs rising, the break-even point for ASIC development is shifting slightly towards higher volumes. This gives a greater market share for FPGAs.

The design process has changed a lot over the years, too. These days design entry at the transistor level is very uncommon. Transistor level modifications are done for low level design optimization, if at all. Languages and tools were introduced which allowed hardware descriptions at a gate and register transfer level (RTL). Design methodologies with appropriate languages and tools were invented to support the new abstraction. This greatly improved design productivity, and since then hardware description languages (HDLs) have always played a very important role in hardware design.

However, current HDLs are not capable of describing all aspects of modern FPGAs. Some FPGA families support modifying the application during its execution. Parts of the application logic are replaced, while other parts remain operational. Current HDLs do not support runtime hardware replacement, since they cannot express this kind of dynamic behavior.

This leads to a low design productivity when such a changing circuit is to be implemented. Many aspects are to be specified manually, e.g. the application dynamics or the control of the logic exchange process. Even worse, simulating such a design prior to its implementation is very difficult.

This thesis proposes one solution to this problem. An extension to an existing HDL is described, which allows expressing dynamic hardware behavior and thus reduces design complexity. The modeling is oriented on the needs of a designer having algorithmic C/C++ descriptions at hand and looking for an easy path to dynamic hardware. Traps and pitfalls in the modeling process are avoided or attenuated to a great extend. Optimization strategies are offered to tune the usage of dynamic resources to the application's needs. Further it is shown, that the resulting language can be simulated and synthesized to a set of RTL descriptions. Finally, a design example is given, where the RTL descriptions are implemented on a FPGA prototyping board using standard hardware development tools.

1.2 State of the Art in Dynamic Partial Reconfiguration

This Section gives a short introduction into the state of the art. A more in-depth analysis can be found in Chapter 2.

In state-of-the-art hardware designs, adaption is usually done by providing all possible logic functions parallel. Only the functionality currently required by the application is visible. The remaining parts are hidden by using multiplexer structures. The control inputs of these multiplexers can be seen as function codes specifying the configuration of a circuit.

Reconfigurable hardware does something comparable. Reconfiguration is the ability to modify the logic function implemented in hardware without changing the system's design. The logic structures of a FPGA are a lot more regular and flexible compared to an *application specific* integrated circuit. One drawback is that it takes data to define the configuration, since a large number of look-up tables, flip-flops, and routing resources need to be controlled. Most often FPGA configuration data is stored in separate memories and loaded onto the device during system initialisation phase. On the other hand, modifying this configuration memory offers a new kind of flexibility: Whatever can be expressed by the flexible FPGA resources may be the next configuration. An ASIC just offers those options that are burned-in during manufacturing.

Dynamic partial reconfiguration is even more challenging. Here, only parts of the circuit are modified and the surrounding logic continues to operate during that change.

Traditional design flows do not support reconfiguration at all. A designer using these features would be left on his own. Implementing such systems requires a lot of manual work. When it comes to combined partial and dynamic (runtime) reconfiguration, the situation is even worse since there is a lot more to take into consideration while designing a system.

There are few FPGAs on the market supporting dynamic partial reconfiguration. The best known product families on the market today are the Xilinx Virtex 2, 4 and 5 families [142, 141, 146]. A lot of research was done to find out ways to exploit the reconfiguration abilities of these chips.

Aside from FPGAs, other kinds of reconfigurable systems were built. Coarse grain reconfigurable machines were proposed that were tailored to specific application domains. Those architectures offer some intermediate way between application specific and fully customizable circuits. The PACT XPP[91] is an example of a commercially available runtime-configurable array of coarse-grain configurable elements.

Reconfiguration is also used to speed-up software processors. Normally, processors are fabricated as ASIC designs tailored to the application of software execution [1]. Processor families differ in their instruction set, speed, energy consumption, form factor and many other aspects. For each application domain, some processors are very suitable and others are not. So, processor vendors aim at broadening the range of applications for which their products can be used. A well-known example is Intel with their MMX instruction set extension improving Pentium's data-stream processing abilities. This can be found elsewhere: PowerPCs have been extended in a similar way by adding an extension called AltiVec, and AMD used an extension titled 3DNow! for their Athlon processors for the same purpose. Reconfiguration offers a different way. Instead of implementing a completely fixed instruction set architecture, some fraction of the microprocessor remains flexible. This flexible part is comparable to a FPGA coupled with the processor core. Depending on the application to be executed, the flexible FPGA part is adapted. A tool framework assists the designer to exploit this flexibility with a minimum of effort. Commercial example are Tensilica's Xtensa processors[128], and the MIPS CorExtend technology[82].

1.3 Goals of this Work

As shown in Section 1.2, reconfiguration is something that can be applied to various types of chips. It blurs the borders of the once fixed separation of application specific circuits and complete general-purpose processors. However, its success was limited so far. The new grades of freedom offered by reconfiguration are also its biggest disadvantage. Whenever design effort is rising to prune the design space, whenever traditional work flows and languages need changes, whenever new complexities are introduced making the results less predictable, whenever new pitfalls are possible, then industry remains very conservative for a good reason. These are problems to be tackled to make dynamic partial reconfiguration successful.

> The main contribution of this work is a platform-independent design flow to exploit dynamic partial reconfiguration on FPGAs. It assumes C/C++ as its entry and delivers RT-level VHDL or SystemC as its results. It covers modeling, simulation and synthesis steps for which it can be that a great extent of tool-assistance can be provided. It shortens design time and improves design quality by reducing manual efforts and avoidance of design pitfalls.

In the following sections, abilities to describe, simulate, analyse and synthesize the framework are presented. There are four major goals for this work:

1. Integration into existing tool-flows. A design flow is a chain of automatic and manual transformations on models. The nature of some manual transformations is to prune the design space by annotating information to a model. Others are to bridge

[1] Examples for exceptions to this rule are software execution functionalities implemented in FPGAs and other technologies.

between the output of one transformation and the input of another. If a design tools fit well into a chain, few or none of such manual transformations of this type are required. The worst case is manual re-coding a model in another language. It is tedious, time-consuming and error-prone labor. Furthermore, it is very difficult to ensure that such a re-coded model matches the source model.

One assumption of the proposed work is that the algorithms to be implemented exist in C++. They are assumed to be integrated into an existing system model or a new system model which needs to be built from scratch. The model is assumed to be expressed using SystemC due to its C++ roots and the constantly increasing industrial use of SystemC. SystemC allows to use every feature offered by C++. The SystemC library will be discussed in more depth in Section 2.4.

Since re-coding is to be avoided, there needs to be a path towards less abstract models. For this very reason, OSSS (see Section 2.5) was introduced by the OFFIS institute. OSSS is a subset of SystemC, omitting those elements for which no synthesis path is known. Giving designer some aid when removing all non-synthesizable elements from a model, new elements were added which fill some of the emerging gaps. Among these are easy ways to introduce inter-process communication, or replace polymorphic pointers. In this step, some elements of the source model map naturally to OSSS replacements, which do express partial reconfiguration. The extension of OSSS with these elements is called OSSS+R.

Once a OSSS+R model is built, it may be simulated, analyzed and synthesized. OSSS already offered simulation and synthesis abilities. The extensions and the analysis support are introduced in this thesis. The synthesis result is VHDL, which can be processed further by standard industrial tools.

2. Short development cycle. Introducing dynamic partial reconfiguration abilities means offering new grades of freedom for the designer. At best, it can be seen as a widened design space which needs to be pruned. This also means that a lot of issues need to be addressed. The process of reconfiguration itself is to be implemented. Configuration control needs to be embedded into the application in a seamless manner. Since reconfigurable parts are unreliable communication partners during modification, non-configurable parts need to be made aware of reconfiguration. The list of issues is even longer when it comes to details.

As stated before, manually modifying the application to exploit dynamic partial reconfiguration is to be avoided. To keep the whole possible design space reachable, little more then some guidance can be offered by tools and languages. One natural answer is to prune the design space in such a manner that a more abstract description is possible.

Restricting the expressiveness of a description language is natural for both hardware and software designers. Hardware designers are used to languages, which allow easy specification of CMOS circuits while not being able to express everything that can be built from transistors, for example domino logic. Software programmers are used to express function calls but usually do not have fine-grain control of processor registers. If desired, they need to use assembler languages, which in turn require more manual work.

In OSSS+R, not every possible use of dynamic partial reconfiguration can be described. OSSS+R restricts the user to some modelling styles. However, reconfigurable systems can be specified quickly using the permitted styles, compared to a manual implementation. In the design of the modeling elements and their syntax, attention was turned to disburden the designer from error-prone tasks. As a consequence, the described system offers a vastly reduced set of possible pitfalls.

A consequence of this pruned design space and tool support is a short development cycle. The modeling elements are oriented on those which can be expected in a C++ model. An OSSS+R model can be quickly rebuilt to a purely static implementation

to asses the benefit and costs of partial reconfiguration. As a consequence, a designer does not need to re-design an OSSS+R implementation completely, if the application turns out to be not suited for dynamic partial reconfiguration. This reduces risks when trying reconfiguration.

3. Design quality. Dynamic partial reconfiguration can be used to make more flexible use of a specific chip area. As a result, the total chip area may be reduced, since a set of tailor-made circuits for multiple purposes may be replaced by a few, more flexible ones.

On the other hand, this flexibility requires some additional control logic. This might in turn result in an increased area overhead. In order to make dynamic partial reconfiguration a feasible approach, the area savings need to be dominant over the additional overhead due to a more complex control logic.

4. Platform independence. Lots of research work was done to utilize the features of specific platforms and vendor tools. As an example, several approaches relied on the Xilinx JBits[56] software. JBits is a Java library allowing bitstream manipulation for Xilinx' first Virtex chip family. Since JBits is discontinued by now, these approaches cannot be applied to current chip families.

Another example is that some research approaches rely on readback capabilities of the FPGA device. Readback is the process of reading a configuration as a bitstream and storing it elsewhere. It is not guaranteed that future devices will support readback abilities. Even worse, for some applications with highly confidential intellectual properties or cryptographic data, it is prohibitive to allow readback to happen.

As a consequence, the proposed approach relies on some general features and not on a specific platform or tool chain. The dependencies are shown in table 1.1.

Feature	Description
Configuration port	The device must offer some interface to upload new configurations.
Deterministic configuration times	It must be possible to determine the duration of configuration processes statically.
*Re*configuration	It must be possible to alter already configured areas by uploading another configuration bitstream.
Partial configuration abilities	It must be possible to have bitstreams altering a fraction of the chip area only.
Dynamic configuration abilities	During a reconfiguration process, those chip areas which are not being reconfigured must be able to continue operating.
Self-configuration abilities	The configuration must be controllable from the application logic itself. Some FPGA families (like Xilinx Spartan III) do not offer an internal configuration port. In such cases, the printed circuit board needs to be designed to permit access to the configuration port using other device I/O pins.

Table 1.1: Platform and tool-chain requirements

Applications of Dynamic Reconfiguration

In the last decade there has been intensive research on which applications benefit most from dynamic reconfiguration. Interestingly, the fundamental kind of benefit achieveable by runtime reconfiguration is not always the same. One can distinguish three major goals:

1. Increase of area utilization. Most applications strive for this goal. By having adaptable hardware resources, one may re-use a resource depending on a demand varying at runtime.

With traditional, static implementation, each functionality possibly needed any time during the circuit's lifetime needs to be available in co-existence to all other functionalities. If the circuit does use a small subset of a large, heterogeneous set of components, many components are unused during runtime. Such a low utilization can be treated as an inefficient use of resources.

Adaptable resources allow re-programming a subcircuit to implement one function on-demand. The amount of circuit space is reduced while maintaining the same functionality, which in turn leads to a better resource utilization. This flexibility comes at a price. While the circuit utilization is required, it demands an external resource providing blueprints for all possible functionalities. In the case of FPGAs, this usually leads to a larger non-volatile memory (e.g. Flash memory) accompanying the FPGA device. The information density of such memories is usually much higher than of FPGA configuration memories. In the end, it is a trade-off between selecting a smaller and cheaper FPGA device with reconfiguration abilities and a larger storage for configuration bitstreams.

Having an FPGA with partial reconfiguration abilities may be an extra bonus. Such devices may control the reconfiguration from within themselves, eliminating the need for external supervising components, e.g. processor units.

Given the goal of improved area utilization, applications suitable for dynamic partial self-reconfiguration can be expected to have the following properties:

1. They require parallelism and performance usually found in hardware, or software implementations are prohibitive due to power consumption. Otherwise, a software implementation is likely easier to implement and maintain.
2. Price or geometrical limitations require using the smallest possible amount of devices on a circuit board. Otherwise, using non-partial reconfiguration would suffice.
3. The application needs a large set of subcircuits, where only a limited set is used at any instant. The upper bound of area utilization is low. Otherwise, the reconfigurable device would not be smaller and cheaper than one used for static implementation.
4. Changes in the used subset of these circuits are made infrequently compared to their duration of use. Otherwise, re-configuration costs in terms of latency and power consumption would dominate the utilization benefits.
5. Latencies while switching circuits can be tolerated or hidden by use of prefetching techniques. Otherwise, functional application constraints could be violated.
6. The extra cost (in terms of price, package geometry or power) for a larger configuration bitstream storage device is totally compensated by using a smaller FPGA for computation tasks. Otherwise, implementing all subcircuitry in parallel would be cheaper.
7. The application must require a reasonable amount of resources. Implementing circuit reconfiguration always includes a considerable overhead due to extra control circuitry. The area saving must also compensate for this methodology-intrinsic overhead.

It is easy to see that dynamic partial self-reconfiguration is applicable for a limited set of applications. It is likely that more and more applications will fit into this category

in the future, since the demand for systems that are simultaneously flexible, cheap, and small is likely to increase. Perhaps the most limiting properties of reconfigurable devices are their price and power consumption. Both can be expected to improve in future, possibly allowing the use of FPGAs in lightweight mobile devices someday.

2. Defect compensation. A less common purpose of dynamic reconfiguration is to compensate for defects. While the reconfiguration process is another source of errors, it may also compensate a class of defects.

One example for such use is FPGA devices that are being re-configured periodically with the very same configuration. This is being done, for example, in the CERN linear particle accelerator [83]. Intense radiation is the source of errors to be compensated for. Particle impacts in the circuit may cause so-called single-event upsets (SEUs), which alter a flip-flops state. If a configuration memory bit is affected, this may alter the FPGA programming and therefore, cause systematic circuit malfunction. By periodically re-programming the configuration memory, these altered flip-flops are corrected.

Other applications would be to perform self-detection of circuit malfunctions and use of reconfiguration to heal defects. In any case, using reconfiguration to compensate defects does not avoid them. The technology rather allows to one take countermeasures. Again, this is a trade-off, since non-reconfigurable devices like Antifuse-based FPGAs or even ASICs provide a better intrinsic protection against some types of errors. Such devices, for example, may be affected by SEUs in data memories only, since the configuration is fixed.

To the authors best knowledge, there is no application of dynamic reconfiguration of FPGAs in safety-critical environments. Even the application in CERN is to make malfunctions less frequent, and not to avoid them. Accepting malfunctions in safety-critical environments is prohibitive.

It is not as easy to formulate application criteria in the case of using dynamic reconfiguration for defect compensation. First, reconfigurability as-such adds to the vulnerability to some kinds of defects. It requires additional hardware resources making manufacturing defects more likely. Configurations are stored in memories, which may be volatile, affected by radiation or similar. The benefits given by the runtime-adaptivity need to compensate for these disadvantages.

The example application from CERN shows that reducing the amount of time for which an error caused by radiation is in effect can be implemented. In principle, this could have been achieved by an ASIC or Antifuse-based FPGA solution, too. In all cases, the application data would still be vulnerable to radiation.

3. Updates in the field. Another application would be to provide circuit updates while the system is in operation [8]. This would be a benefit in remote or inaccessible environments. Examples range from digital radio base band stations to electronically equipped buoys. By transmitting an updated circuit over the air and automatically self-updating a circuit, no manual service would be required.

Applications to benefit from dynamically updating systems in the field can be expected to have these properties:

1. They demand a degree of parallelism and speed typically found in hardware solutions. Otherwise, a software solution would satisfy the needs.
2. The application must not be switched off during reconfiguration. Otherwise, self-updating would be impossible. In turn that would require manual interaction by a service person.
3. They must be distributed, inaccessible, remote or even difficult to modify physically (e.g. sealed). In other words, it must be costly to apply maintenance updates. Otherwise, the benefit of not sending out service persons would be small.

OSSS+R. OSSS+R strives for the goal presented first, the increase in area-utilization. It may be extended to support the second and third goal, but that is beyond the scope of this thesis. Regardless of which goal is to be achieved by dynamic partial reconfiguration, it is mandatory that the implementation process is quick, easy to learn and safe. Otherwise, the potential benefits of dynamic partial reconfiguration would even need to outweigh a time-consuming, difficult and, error-prone design process. OSSS+R aims at lowering these risks for projects.

A Brief Overview

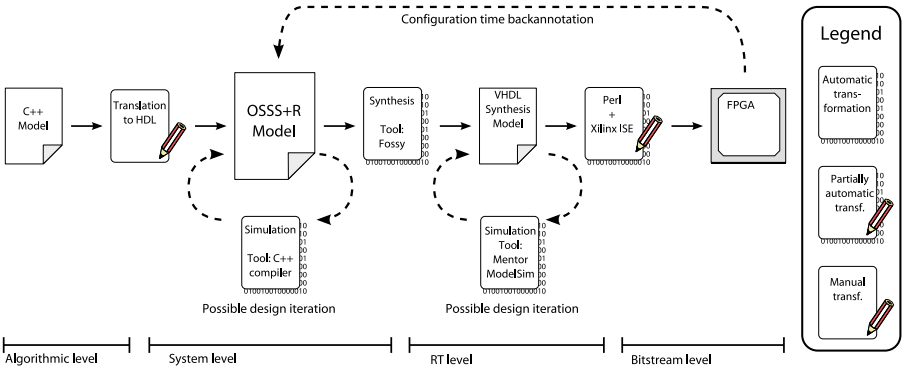

Figure 1.2: Simplified tool flow for OSSS+R

The proposed approach, named OSSS+R, is structured as shown in Figure 1.2. The design entry is a set of C++ algorithms that are to be embedded into an existing SystemC model or the foundation for a SystemC model that is to be built from scratch. First, some C++ aspects like pointers, polymorphism, or dynamic memory management need to be replaced by synthesizable counterparts as offered by OSSS+R. This has to be done manually. The resulting model can be simulated and shows the impact of partial reconfiguration. If it shows the intended behavior, a synthesis tool transforms the model into register transfer level code. The resulting code may be VHDL or SystemC (not shown in Figure 1.2). It can be simulated and further synthesized using standard industrial tools. Once bitstreams are generated by FPGA vendor tools, the correct duration of the configuration processes can be calculated. These values can be back-annotated to the OSSS+R model making a high-level but accurate simulation possible.

Evaluation Criteria

The evaluation criteria for this work is specified by the four goals as described in Section 1.3.

The tool flow integration can be assessed by processing an example and identifying the links to the traditional tool-chain. The second goal is a short development cycle. This cannot be judged easily. To have reliable figures, two teams would be needed to implement the same set of applications. Both teams would need the same skills and the same application knowledge. This is difficult to accomplish in the scope of this work. Goal three is the design quality. To implement the examples presented in this work, Xilinx vendor tools are used. They offer figures for a number of used look-up tables and flip-flops for a design. These figures do not give perfect answers, since there is no support for traceability. However, the area impact of the proposed approach

can be sketched. The last goal is platform independence. Table 1.1 lists the platform requirements which can be discussed in light of the proposed approach.

1.4 Outline of this Work

The remaining work is structured as follows. Chapter 2 gives an overview of the state of the art. The first focus is on SystemC as the dominant system description language and OSSS as a synthesis subset. A second focus lies on dynamic partial reconfiguration. Chapter 3 gives an overview on related work. Since there are numerous approaches having reconfiguration as their topic, it focuses on the most similar ones. These are especially those which also aim at fine-grain reconfigurable devices and express the reconfiguration process in their design entry. Chapter 4 introduces the proposed modeling of reconfigurable systems. It also discusses possible optimizations to improve the usage of reconfigurable areas. Next, Chapter 5 explains the syntax used for modeling, and its integration into SystemC and the simulation abilities. It is followed by Chapter 6, which covers the transformation to a register-transfer level model. Chapter 7 explains the experiments to assess the approach. Here, the approach is evaluated with respect to the criteria introduced before. Finally, Chapter 8 gives an summary and an outlook for possible future work.

Chapter 2

State of the Art

2.1 System Design

The complexity of designs is increasing over the years. An ever-increasing demand for more integrated and more powerful systems and continuously improved manufacturing abilities are driving this trend. The more complex a system is, the more challenging the design process gets. People need to use abstraction techniques to keep things under control. A initial specification is often transformed into abstract models first. These models are analyzed using formal methodologies or simulation to seek design flaws. Acceptable models are then iteratively enriched with more details to form a implementable model. Ideally, such a model can be simulated at each step of refinement. A good synthesis flow allows the designer to re-use as much as possible of a model in each refinement step. This shortens the design process and reduces the likeliness of mistakes being made.

System design is a very ambitious goal in this respect. To capture a complete system, like car electronics or a flight control system of an airplane in great detail is somewhat in between challenging and infeasible. A model's size, its heterogeneity, large timescales, and a complex and dynamic environment complicate this task. Again, abstraction is used to simplify models.

One key requirement for a design language stems from these considerations: It must support multiple levels of abstraction. A very sketchy initial model is to be refined iteratively without leaving the scope of the language. The C++ library SystemC[90] is increasingly popular and aims at fulfilling these goals. It was standardized as a library with IEEE 1666-2005.

A typical design entry of a SystemC-based design flow is a set of algorithms described using C/C++. These are the foundation for a new structural model or are to be embedded into an existing one. Some C/C++ elements are preserved as software components and mapped to general purpose processors (GPPs) or digital signal processors (DSPs). Others are to be implemented in hardware. To map a C/C++ description to a hardware target requires some severe modifications. The key issues in this process are:

Parallelism. C and C++ descriptions are purely sequential by definition. The language standard defines, which operations are to be executed in which order. However, in some cases it can be proved that the sequence of operations can be modified without impact on the program semantics. To speed up software execution on modern processors, this is used very excessively. Operations which can be executed independently of each other may be executed in parallel on processors. It is the compiler's task to arrange processor instructions in such an order that the degree of parallelism during execution can be maximized. Modern processors dynamically calculate runtime dependencies of instructions and parallelize their execution. The techniques in this field

have become very sophisticated over the decades, ranging from compile-time support for parallelism (e.g. for Intel Itanium processors) to speculative execution. However, the possible degree of instruction-level parallelism is limited. To further boost processor performance, the software programmer has to express parallelism on an additional level. This thread-level performance is a very active field of research these days with a strong industrial relevance.

In contrast to this, hardware offers a great degree of parallelism by nature. Hardware description languages support concurrency as an intrinsic feature. As an example, VHDL and Verilog offer thread level parallelism by providing concurrent processes. Data dependencies determine the execution order within one block of combinatorial logic. Concurrent assignments can be used to explicitly express the independence of assignments in one combinatorial logic block.

In summary, transforming a sequential software description into a hardware description requires explicitly introducing parallelism. Concurrent algorithmic fractions need to be identified and expressed. SystemC as a design specification language offers elements for this task: Clocked threads offer implicitly described state machines, combinatorial methods offer explicit state machines and signals allow concurrent assignments.

Communication. Splitting up a sequential description into a set of parallel components does not make these components independent of each other. The new components need to communicate with each other to fulfill the original task. Consider a blocking C function call, which is to be transformed into a non-blocking call, where the caller continues its calculation until the result of that function call is to be consumed. The designer needs to give an answer on how the call is to be implemented, how the results are received, how the completion of the function can be detected and various other questions.

Traditional hardware design languages offer signal-based communication only, which is on a much lower level of abstraction. SystemC also allows transaction level modeling (TLM), which permits method-based communication.

Scheduling. C and C++ descriptions are untimed. Software programs have no notion of time since the execution speed is determined by the processor and the connected components only. Timers and process priorities as offered by operating systems are not part of the C and C++ languages.

Hardware however is typically clocked. A sequence of fixed-length clock cycles requires to schedule operations in a balanced way. Unbalanced scheduling leads to an increased clock cycle length, which in turn causes decreased system performance. To maximise concurrency, there should be as little as possible dependencies among operations scheduled to the same cycle.

A design language needs to be able to express such a schedule. SystemC offers separating states of an implicitly described state machine using so-called "wait statements". Instructions placed in-between two wait statements are scheduled to the same clock cycle. Such implicit state machines are closer to an untimed C/C++ description compared to explicit state machines.

Datatypes. Software languages are typically constructed to abstract from processor capabilities. For example, processors operate on data words. Modern processors typically operate on bitwidths of 8, 16, 32 or even 64 bits width. Single-bit accesses are typically unsupported assuch. Modifying a single bit in memory requires fetching a word, modifying a bit inside this word and finally writing the complete word. Consequently, C and C++ offer datatypes of word sizes.

Hardware implementations consist of functional units performing bit-level optimized operations. As an example, if a computation yields a result of 16 bit width

having 4 bits of fixed value, the resulting logic will compute 12 bits only. The same applies to inputs: Having some bits of an input being fixed will lead to optimized operations which exploit this additional knowledge and leave the fixed inputs unconnected. A direct consequence of these properties is, that hardware datatypes often do not match those offered by software programming languages. Hardware is bit-oriented and not word-oriented as software.

SystemC allows creating types of various lengths and signedness, fixed-point types and floating-point types.

Von-Neumann model and dynamic memories Normal software programs are written and compiled with the assumption of uniform memory. All data is stored in a logical memory where the access latency and bandwidth is independent of the location inside the memory. In real implementations this assumption is partially incorrect, since caches lead to different latencies and bandwidths depending on data alignment, data ordering and data access history. This is typically being ignored in software programs. The benefit of assuming an uniform memory is a greatly reduced complexity when writing and compiling software.

Since hardware has a much higher degree of concurrency and operations are not executed strictly sequential, memory accesses are becoming concurrent, too. As a consequence, having a single uniform memory to store data is no longer feasible. In custom hardware descriptions, small memories are distributed over the circuit. They are implemented as latches and flip-flops embedded in the circuit's datapath. Large memories, however, may be instantiated explicitly as separate blocks, for example as DRAM memory. In case of FPGAs, chip families often offer multiple dedicated on-chip memories for this purpose.

Another convenient assumption of many software programs is to have an unlimited memory space. An operating system offers functions to allocate and free memory blocks during runtime. Memory is allocated as a virtual space and then mapped into physical memory on demand. This is completely transparent to the software application. Exceeding physical memory limits automatically leads to the use of swap space on slower but larger storage. Even fragmentation issues are hidden from the application.

As said before, regular hardware memories are distributed over the circuit area. Their size needs to be determined statically at design time. These memories are dedicated to contain the values of variables or a arrays of fixed size. Implementing a complex memory management supporting dynamic allocation and destruction is infeasible due to the area and timing overhead. Memory management operations must be stripped off the C/C++ model and are to be replaced by a static memory management.

2.2 C++

The approach presented in this thesis is based on SystemC. Since C++ is a software programming language and SystemC is a C++ software library used to describe hardware, some peculiarities of C++ are presented in this section to clarify relevant properties of this language.

2.2.1 Pointers and Instances

C++ heavily relies on a von-Neumann-architecture. It is helpful to envision the implications of this underlying concept in order to understand polymorphism in that programming language. C++ assumes a uniform addressable memory, where all locations can be accessed using the same operations and with the same effort. Todays implementations of such machines usually include complex memory hierarchies using caches but this is not reflected in the programming model itself. The elements of this uniform

memory are enumerated. These numbers are used to address data inside the memory. Essentially, pointers are such addresses of memory elements paired with an type information on the data being stored inside the memory. Borrowed from its *C* roots, C++ knows certain simple types (like integer or character types), as well as aggregates. Such aggregated types are *vectors* (sequences of identical types), *structs* (ordered collection of varying types), or *unions* (collection of alternative interpretations of shared data). If structs are paired with operations being defined on them, they are called *classes*. Instances of such classes are called *objects*. Pointers are simple types, which represent the address of the datatype instance. Due to this construction, pointer types include the type of the instance being referred to as an intrinsic part[1].

For brevity of this thesis, the term *pointer* is used in the sense of pointing to an object from now on. C++ does provide other types of pointers but since those types are not part of any inheritance tree by definition, they are not relevant when discussing polymorphism.

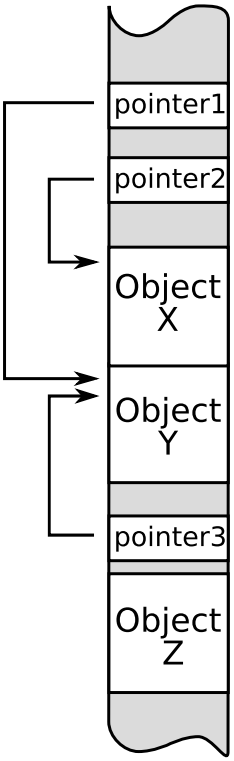

Figure 2.1: Memory layout in a Von Neumann architecture

Figure 2.1 shows a memory layout excerpt for an artificial example. Three pointers are shown (`pointer1` to `pointer3` as well as three objects (`X` to `Z`). In C++, one can make the following observations:

1. Pointers always refer to exactly one object at a time.
2. Pointers may point to different objects at runtime. Example: `pointer1` may be modified to refer to `Z`.
3. Multiple pointers may refer to the same object. Example: Both `pointer1` and `pointer2` refer to object `X`.
4. There may be objects which are not being pointed at. Example: No pointer refers to `Z`.

[1] C++ knows just one exception to this rule, being the *void* pointer where the type of data being pointed at is treated as unknown by purpose.

5. There may be calculations involving pointers. Example: Since object `Y` directly follows object `X`, `pointer2 + 1` refers to `Y`, if the runtime types of `X` and `Y` match the destination type of `pointer2`.
6. Both pointers and objects may be dynamically created and destructed.
7. In ill-formed programs, pointers may refer to non-object locations.
8. A special address `NULL` is reserved referring to no object at all.

Since there are very few restrictions in this system, it is very flexible to use by the programmer. This comes at a cost. The biggest drawback is, that it is nearly impossible to modify the underlying memory model. Whenever the efficiency of the memory model is to be increased, some of the flexibility needs to be abandoned. There are tools on the market which try to analyse, which dependencies, which dynamic behavior and which unique C++ features[2] are used in a given program. If certain usage patterns can be proven by those tools, optimizations may be applied. As an example, provided that no self-modifying code exists, program instructions and data can be accessed using separate memory hierarchies (Harvard architecture). It is not possible to write a tool which performs all possible optimizations on an arbitrary program, due to the fact that this would require the tool to understanding program semantics.

For a hardware implementation, it is advisable to modify the C++ memory model. Hardware implementations may show a massive and very fine-grained parallelism. This is not possible when accessing data trough a Von Neumann bottleneck. As a consequence, not all abilities of C++ pointers are usable for modeling reconfigurable hardware in the system proposed in this thesis.

2.2.2 Polymorphism in C++

As explained before, C++ pointers are always associated with a type. Pointers to objects are therefore associated a with class type. These class types may be in a so-called "is-a" relation to another, expressed by *inheritance*. This paragraph introduces a concept called *polymorphism*.

> **Definition 1** - ***Polymorphism (C++)***:
> *Polymorphism is the separation of pointer destination type (static type) and the type of the object being pointed at (dynamic type). The static and dynamic type need to be in a is-a relation, that means, the dynamic type needs to be identical to the static type or inherited from it.*

To completely understand this relation, it is advisable to be familiar with further literature about *object-orientation* [71, 25]. In particular classes, inheritance, overwriting of operators, and virtual member functions are important concepts. The following section gives an overview of polymorphism in respect to C++.

As an example, Figure 2.2 depicts a minimalistic class tree in UML[4] notation containing a base class `A` and three derived classes `B`, `C`, and `D`. There are three functions, `foo()`, `bar()`, and `extra()` being declared in some of these classes. Only `foo()` and `bar()` are declared in the base class `A`, where `foo()` is declared virtual and `bar()` is non-virtual. All three are inherited in `B`, whereas `B` overloads them `virtual` and `C` overloads them non-`virtual`. Since UML does not have a fixed notation for non-virtual member functions, it is indicated by the stereotype `non-virtual` in the class diagram. This way, all possible combinations of method definitions (not done, virtual and non-virtual) in base and derived classes are given.

C++ provides polymorphism though pointers, for example as shown below:

```
1 A * poly;
```

[2] Like pointer arithmetic, for example.

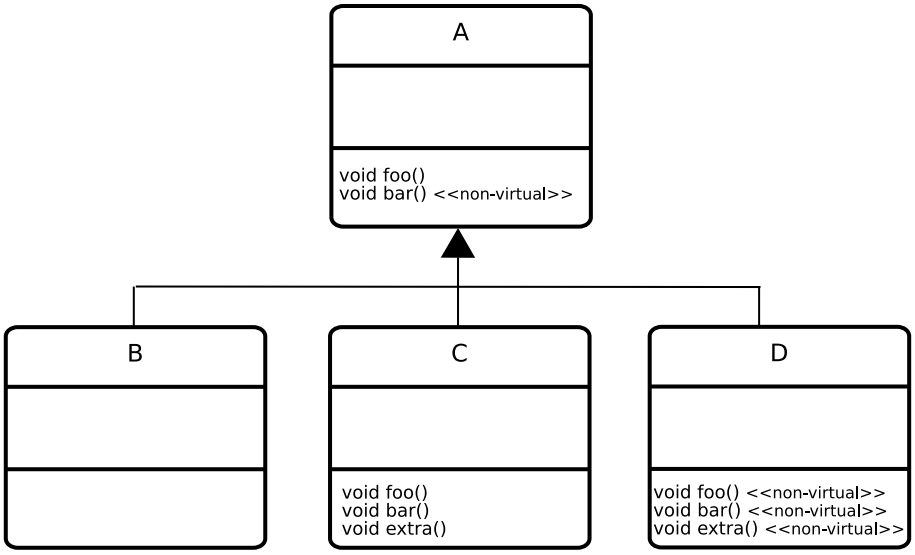

Figure 2.2: Simple class diagram

```
2  poly = new A();
3  poly->foo();    // calls A::foo()
4  poly->bar();    // calls A::bar()
5  poly->extra(); // not allowed, since not in A
6  delete poly;
7
8  poly = new B();
9  poly->foo();    // calls A::foo()
10 poly->bar();    // calls A::bar()
11 poly->extra(); // not allowed, since not in A
12 delete poly;
13
14 poly = new C();
15 poly->foo();    // calls C::foo()
16 poly->bar();    // calls A::bar()
17 poly->extra(); // not allowed, since not in A
18 delete poly;
19
20 poly = new D();
21 poly->foo();    // calls D::foo()
22 poly->bar();    // calls A::bar()
23 poly->extra(); // not allowed, since not in A
24 delete poly;
```

It is being distinguished between static and dynamic types. Strictly speaking, the static type of `poly` is `A*`. It is a pointer to an instance of class `A`. The dynamic types are the class-types of those instances, which `poly` may point at: each of `A`, `B`, `C`, and `D`. This is due to the nature of class inheritance defining an *is-a* relation on classes. Objects of type `B` can be treated as objects of type `A`. A static type can be determined at compile-time. The compiler can rely on the is-a relation of static and dynamic type.

The inheritance relation between static type and dynamic type adheres to Liskov's substitution principle[72]. This principle ensures that no relevant property of a class is modified or removed when deriving classes. Applied to C++ polymorphism, the

dynamic type is guaranteed to provide all members as provided by the static type.

Object manipulations that are possible trough a polymorphic pointer are limited to the properties which the C++ compiler can guarantee to be available by using this pointer. In other words, the static type of the pointer determines a fixed interface available for use. Additional properties of the dynamic type are inaccessible through the polymophic pointer. This can be observed in the given code example. Calls to `foo()` and `bar()` are permitted, regardless of which object `A` is pointing at. No calls to `extra()` are allowed, since the method interface of `A` does not contain such a method.

Another observation from the given code sequence is that the method implementations of varying classes used for execution depend on the function signature and the dynamic type. The difference in the function signature (method identifier and formal parameters) in this example is only the name. The only difference between `foo()` and `bar()` is that `foo()` is defined as being *virtual* and `bar()` is not. A method being virtual in the *static* type of the polymorphic pointer is resolved at runtime whereas a non-virtual method is resolved at compile time. Compile-time resolution requires that the method implementation to be called needs to be independent of the runtime type. This requires calling `A::bar()`. Since `foo()` is declared as virtual in `A`, the compiler resolves it at runtime. Since both `C` and `D` provide their own implementations, those are used. Class `B` does not provide a specialized one, so the class tree is searched upwards to the root for an implementation of `foo()`, ending at `A`. The bottom line is that the virtuality of a method in the pointer's static type determines, which function is being called. The virtuality of that method in the runtime type is not relevant.

It is not always necessary to put the *virtual* keyword in a method's definition to make it virtual. It is sufficient if the method was declared virtual by a parent class. In the given example, `D::foo()` is virtual due to `A::foo()` being virtual. This effect does work only downwards in the inheritance tree, not upwards: `A::bar()` is not made virtual by `C::bar()`.

2.3 Hardware Description using C++ and SystemC

Value and Entity Objects

Since SystemC is a hardware description library, it describes both algorithm and structure of a design. The structure of the design is described in terms of a hierarchy of modules and their communication infrastructure (using signals, channels and ports). The types of these elements have significantly different characteristics than ordinary data types.

All types defined in C++ are value types. They may appear as local variables or parameters in methods and functions, and their value can be copied. In contrast to these value types, SystemC also provides entity types, which are definable by the absence of these properties. Examples for entity objects are signals, ports, or modules. One may read a value from a port, but cannot copy the port itself. One may use signal-based communication to read the state of a module, but the module itself cannot be copied. Also, it is impossible to use signals, ports, and modules as local members of a C function. Entity objects have an unlimited lifetime, whereas data objects may have limited lifetime.

In SystemC, the topmost entity objects are modules instantiated by the `sc_main` function. Being entity objects, they may contain further entity objects as well as data objects. Data objects, however, may not contain entity objects. For example, a local variable may not contain a SystemC module. An entity object might have a value which can be read, but reading it yields a value object, not a second entity object.

Table 2.1 summarizes the properties.

Property	Instance of data type	Entity type
Copying	Yes	No
Lifetime of instances	Limited or unlimited	Unlimited
May contain entities	No	Yes

Table 2.1: Value and entity types

Formal Definitions

This section describes some formal declarations about runtime reconfiguration. Using multiple reconfigurable devices in one system is possible in the proposed approach. Since no interaction between the reconfiguration infrastructure of two or more devices is modelled, all devices are treated as being independent. For that reason no explicit multi-device modeling is introduced in the given formalism.

First, a formal way to specify class relations is required. The set of user-defined class types is given by $\mathcal{T} := \{t_0, ...t_n\}, n \in \mathbb{N}$ being an unique class id.

There is an inheritance relation $\lhd : \mathcal{T} \times \mathcal{T}$:

$$t_a \lhd t_b \Leftrightarrow t_b \text{ is derived from } t_a \qquad \text{(base class)}$$
$$t_a \lhd^1 t_b \Leftrightarrow (t_a \lhd t_b) \wedge \nexists t_c : (t_a \lhd t_c) \wedge (t_c \lhd t_b) \qquad \text{(direct base)}$$
$$t_a \lhd^+ t_b \Leftrightarrow (t_a \lhd t_b) \wedge (t_a \not\lhd^1 t_b) \qquad \text{(indirect base)}$$
$$t_a \lhd^* t_b \Leftrightarrow (t_a \lhd t_b) \vee (t_a = t_b) \qquad \text{(base or equality)}$$

With the usual notation for negative relationship: $t_a \not\lhd t_b \Leftrightarrow \neg(t_a \lhd t_b)$. The relations $\not\lhd^1$, $\not\lhd^+$ and $\not\lhd^*$ are defined accordingly. Additionally, $\bowtie$ relation defines the membership of a class tree:
$t_a \bowtie t_b \Leftrightarrow (t_a \lhd^* t_b) \vee (t_b \lhd t_a)$ and $t_a \not\bowtie t_b \Leftrightarrow \neg(t_a \bowtie t_b)$.

Polymorphism and Resource Allocation

Although polymorphism is tied to the use of pointers in C++, it is not the only way to implement it. Figure 2.3 illustrates two extreme situations: On the left hand side a completely flexible resource management is shown, illustrating a software implementation for a Von-Neumann architecture. It depicts a single resource pool, which is given by the memory address space. All pointers are located there, as well as, all objects being pointed at. In the given example, there are two pointers, `A` and `B`, being the polymorphic elements. Two objects are shown, one of type `X` and one of type `Y`. Both objects are of types compatible with `A` (Let type of A be `Base`, then `Base` $\lhd^*$ X and `Base` $\lhd^*$ `Y`) with Object `Y` requiring more space than `X`. The runtime type association is given by a small introductory memory fragment, the *type tag*. It serves as a type identification of the following resource fragment. This is a situation which can be found in real C++ programs.

The right hand side shows a strongly restricted implementation, which permits parallel implementation due to strictly separated resources. The polymorphic elements do not point at a resource location. Instead, they *are* resource locations which may contain different objects at runtime. For that reason, they are sized to fit the largest possible object. In the given example, the space of element `A` has enough space to implement `Y`. It does not shrink at runtime, if `X` is implemented. Each polymorphic element is an isolated resource and could use its own set of addresses. Therefore, one may implement each of these with disjunctive physical resources. This comes at the cost of flexibility:

- Objects need to be statically associated with a polymorphic element, for example, no modification of polymorphic element affects another one.

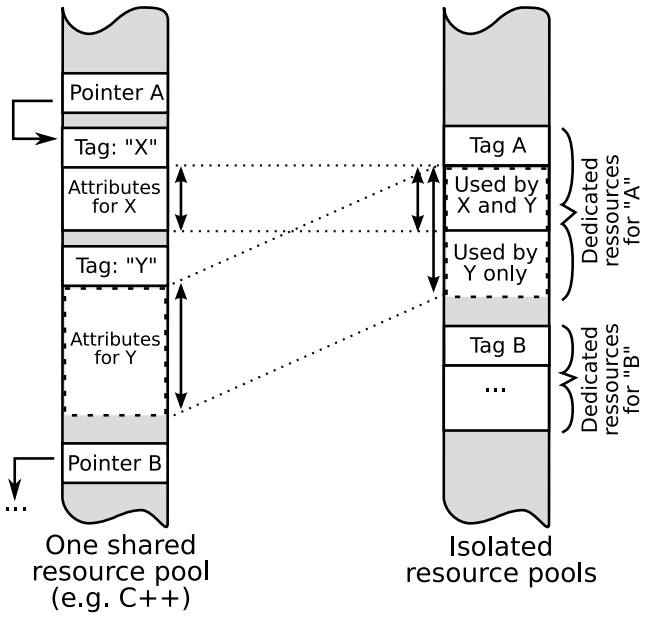

Figure 2.3: Polymorphism and resource allocation

- Unused resource parts of one polymorphic element cannot be used for other purposes.
- There is no expression of a "no object", like a C++ `NULL`-pointer.
- It is impossible to address invalid resources. This can be considered as an advantage.

Since a software implementation typically runs on Von-Neumann machines, software languages like C++ support the more flexible memory model better. The restricted model offers more parallelism, which in turn is not exploitable on these machines. If the restricted model is really wanted, the programmer needs to manually implement it. This requires more manual programming work but is strongly possible in C++.

2.4 SystemC

General

In late 2005, the SystemC library was standardized as IEEE 1666-2005. The Open SystemC Initiative[89] offers a free reference implementation of this standard. From a technical point of view, SystemC models are C++ programs that can be compiled and linked against this library. The library provides primitives, which can be used to model hardware. When linking the program, a discrete-event simulation kernel is added which is comparable to those of Verilog or VHDL simulators. The executable file is both, a model with a built-in simulator.

Executing a SystemC-model is done in two phases: elaboration and simulation. During elaboration, the simulation environment is set up. The simulation kernel sets up internal datastructures and initializes the model to be simulated. Instances in the design hierarchy are constructed, sensitivity lists are set up, ports and signal connections are established and processes are added to the execution list. A SystemC program that compiles well might fail at elaboration. The SystemC kernel displays additional messages if invalid or incomplete settings are detected during elaboration phase.

After elaboration was completed, the simulation is started. The discrete-event kernel schedules invocations of tasks or concurrent assignments according to their sensitivity list. Like industrial VHDL simulators, it uses delta-cycle semantics for this task.

The following code shows a fragment of a hardware module to help illustrate SystemC language elements.

```
SC_MODULE( Example )
{
  public:
  // ports
  sc_in< bool >           pi_clock;
  sc_in< bool >           pi_reset;
  sc_in< sc_uint< 16 > > pi_data;
  ...
  // signals
  sc_signal< sc_uint< 16 > > s_data;
  ...

  private:
  // processes
  void myProc()
  {
    s_data.write( pi_data.read() );
    wait();
    ...
  }

  SC_CTOR( Example )
  {
    SC_CTHREAD( myProc, pi_clock.pos() );
    reset_signal_is( pi_reset, true );
  }
};
```

Hierarchy

Design hierarchy is built of modules. Technically, module definitions are classes inherited from a SystemC library class. Since classes are types, a class definition is comparable to a VHDL architecture, being the blue-print of design instances. Instantiating such an architecture yields a module object, which resembles a VHDL entity.

A module without any processes in its body can be seen as a module interface. They can be used to derived module implementation classes containing the logic to be implemented and possibly further sub-modules. Separating module interface and implementation resembles the separation between component and architecture. Since the instantiation of modules must be done using the implementation anyway, separate interface classes are often omitted.

A SystemC module may contain anything that can be a member of a C++ class. Since the primary purpose of SystemC is modeling and simulation, this makes perfect sense. When it comes to hardware synthesis, the list of potential module members is considerably shorter:

- Processes
- Method members
- Data members
- Further modules as sub-modules
- Signals
- Signal-ports for communication with parent modules
- Channels and interface-ports to channels

Events, signal-ports and signals. The most explicit way to express communication is using signals. Signals represent a set of hardware wires and carry logic values. In SystemC, such signals are declared using the `sc_signal<T>` template, where `T` is the datatype of the signal. The datatype determines the width of the signal in the number of wires and the interpretation of the signal's values. Signals cross hierarchy borders (module borders) through signal-ports. These ports are declared in a similar way using `sc_port<T>`. Changes in signal values cause *events*, which are used to trigger computation.

Processes and methods The functionality inside modules is expressed using processes. SystemC provides different kinds of processes for different purposes. The key question for the choice of process kind is the question of how to express the processes' state and its transitions. The state refers to memory elements storing the results of the last completed step of computation. Transitions are the calculations of the next-step logic.

`SC_METHOD` is used to declare processes that are sensitive to a list of signals. To be precise, the processes are sensitive to the events caused by value changes on signals. A truth table, for example, represents combinatorial logic. It is required to recompute the output of such a logic block only due to changing inputs. As long as inputs are unchanged, the logic output will stay unchanged, too. Performing computations on demand accelerates simulation. The sensitivity list of a process implementing a truth table must include all signals that are consumed during calculation. This is not enforced by SystemC, so it is possible to model processes, which are not directly implementable in hardware.

A special, but useful case of a process with an incomplete sensitivity list is a process only being sensitive to an edge of a clock signal. For such processes, it is clear what the hardware implementation would be: A process with an explicit state machine. The process state memory is given by all signals that are both read and written by the process. If a signal is read-only, it is an input, and write-only signals are outputs. The process body itself is executed whenever the clock changes (high or low transition) and describes the computation within one clock cycle. Signals are updated at the following clock edge the process is sensitive to. The process expresses all transitions of a finite state machine that makes one transition per clock cycle. Such a process is said to have an *explicit state machine*, since the state encoding and transitions are explicitly described.

C and C++ software implementations do not have a notion of time. There is no such thing as a clock boundary to break up computation into pieces. Software programmers are used to implicit state encoding. A processes' progress is measured in advancement inside the control flow. Software programmers rather ask "Did the process enter its inner loop?" then "Does the instruction pointer refer to a position inside the inner loop in clock cycle 428?". The state encoding is not made explicit and the exact clock cycle is not measured.

While (synchronous) hardware requires splitting computations into clock cycles, it may well allow an abstraction from the state encoding. SystemC provides clocked threads (`SC_CTHREAD`s) in which the process body is broken into clock cycles using `wait()` statements. If the thread of control reaches a wait-statement, the computation for the current clock cycle is completed. The advantage of this modeling style is that sequences of statements may be used that span over several clock cycles. This is increasingly convenient the more loops are used in the algorithm specification. Since the computations inside a clock cycle do not range from the very beginning to the very end of the processes' body (as it is the case with `SC_METHOD` processes) and due to the fact that the state encoding is not made explicitly visible by a set of signals, such processes are said to have *implicit state machines*.

SystemC offers a third kind of processes. `SC_THREAD`s are more general threads,

that also allow modeling using implicit state machines. Note, that omitting the letter `C` indicates that they are not necessarily clocked. However, they are sensitive to a set of signals rather than only a single clock signal edge. The sensitivity may even change during execution, depending on the thread of control. For these threads, it is an open question what a suitable hardware implementation would be.

SystemC processes may well invoke C++ member functions. Processes are implemented as C++ member functions in SystemC modules and these modules are just classes themselves. But not all member functions in modules are processes! Non-process methods can be invoked by processes, given they are suitable for the type of process. For example, a clocked thread may execute member functions containing wait-statements, a non-clocked thread may contain wait-for-event statements and a pure `SC_METHOD` process may contain none of them. Member functions invoked by processes can be thought of being inlined into the calling process. If the statement is valid in the caller's body, the invocation is permitted. Since this inlining is recursive, the relation becomes transitive over the call-tree.

Interface-ports and channels. Signals and signal-ports are communication primitives with very little abstraction. They do not implement a *rendezvous* concept or a message-passing concept but provide the basis to implement both styles. Making handshaking explicit is not desirable in abstract modeling.

A solution is provided by *channels*. Channels provide a method interface to be used by processes. This method interface is made accessible through so-called interface-ports. The processes invoke methods on these ports, which are then executed inside the channel. A channel models an abstract communication infrastructure, which can be anything ranging from a point-to-point connection over local buses to on-chip networks. Channels may even contain processes themselves.

SystemC distinguishes between *hierarchical* and *primitive* channels. While hierarchical channels offer the full range of flexibility, primitive channels are more restricted. Primitive channels only have a request-and-update semantics. Updates are requested during simulation but are made visible on progress of time only (not during delta-cycle advances). Signals are examples of such primitive channels, offering read and write methods only. Primitive channels have no internal hierarchy. They do not contain processes or even other channels. FIFOs or mutexes are other examples, which can be implemented using primitive channels. Primitive channels have clear semantics for hardware. They can be automatically mapped to primitives of the synthesizable subset of other hardware description languages, e.g. Verilog. This is not the case for hierarchical channels. While these can be manually decomposed into a hierarchy of modules and primitive channels, the transformation cannot always be done automatically by a tool.

Data members. Since a module definition essentially is a C++ class, it may contain non-signal data members. Data members are like variables placed in the module itself and not in a process. They represent memories which can be implemented in the hardware. There are two cases to be distinguished:

If a data member is used by one process only, it is exclusive to this process. Therefore it can be inlined into that process as a local variable. This would be feasible, if the data member is initialized by the processes' reset code anyway. Then inlining the member would preserve the model's semantics. Non-initialized data members are better explicitly specified as signals to clarify their process-independent existence.

The second case is shared use of data members by multiple processes. In case of signals a clear semantics is defined, since they forbid multiple writers. Unfortunately there is no such multiple-writer check for plain data members of modules. Depending on the order in which a simulator invokes processes, data hazards (both read and write hazards) are possible. There is no mechanism to detect these.

As a general rule, data members should be manually inlined into processes or manually translated to signals. In Section 2.5.4, a possible solution to the problem of multiple writers is described.

2.5 OSSS

2.5.1 Synthesizable Subset

OSSS is an abbreviation for *Oldenburg System Synthesis Subset*. As indicated by the term *synthesis subset*, it is a subset of SystemC that excludes all elements for which no synthesis semantics is known. This restriction to a subset of library elements is conformant to the "SystemC Synthesizable Subset Draft.1.1.18" document provided by the Open SystemC Initiative [89]. For example, it rules out pointers that are not statically resolvable. The lack of pointer support will be of importance in Chapter 4. With this restriction, C++ polymorphism cannot be synthesized, since it heavily relies on pointers. Section 2.5.3 describes a replacement for C++ pointer-based polymorphism as offered by OSSS.

Dynamic memory management using `new` and `delete` operators also is forbidden.

The subset also excludes data members inside a module that are accessed by multiple processes. If multiple processes perform accesses to a variable, the access order is undefined by SystemC. Even if a SystemC simulator is deterministic, a different simulator or even a different version of the same simulator may use a different order of accesses. This is likely to have impact on both simulation results and synthesis. Again, OSSS offers a replacement, which is described in Section 2.5.4.

Channels are not mentioned in the SystemC Synthesisable Subset Draft. Instead, signals are said to be supported. So, these primitive channels are the only ones which are considered synthesizable.

2.5.2 Hardware Implementation of Object Oriented Descriptions

In the ODETTE project [5, 53], the OFFIS Institute developed a synthesis methodology for object-oriented SystemC descriptions. It is the foundation of the reconfiguration synthesis as presented in this work. Although completely explaining the whole implementation strategy would exceed the available space in this thesis, it is helpful to outline some principles.

Member function lowering. In procedural languages, functions operate on a given set of arguments and on global variables. Object-oriented languages like C++ also know *member functions*. These are defined by the user of a class type. Each member function operates on an object[3] which may be modified, if the function is not declared to be `const`. In C++, this is done by passing a pointer to the object as an extra parameter to the member function. This extra parameter is invisible to the programmer in the argument list. Inside the member function, it can be accessed using the identifier `this`. All member attributes of the class (both functions and data attributes) are automatically available to the programmer without prefixing with `this->`.

This extra `this` parameter is made explicit in the function lowering step. It is inserted as an additional first function parameter to each member function. The parameter type is of the class type which the member function is associated to. Next, the member functions are removed from the class definition. All uses of these member functions are replaced by the use of the freshly created, lowered functions. Only data members are left inside the class, turning it into an equivalent to a C `struct`.

[3] Ignoring `static` member functions here.

In C++, the `this` parameter is passed as a pointer to the object. The lowered functions uses a copy-in-copy-out call semantics instead and passes the object data to the function. In case of a `const` member function, a call-by-value semantics is sufficient. This way it is avoided to synthesize pointers for the `this`-parameter.

Member function calls are then replaced by appropriate function calls that operate on the structs that once were classes. Virtual member function calls are resolved to non-virtual function calls, that means, the runtime type of the object is hard-coded into the design. After member function lowering, all objects and methods are transformed into C structures and functions, respectively.

Function inlining. Apart from the complexity of operation, there is no difference between executing a built-in operation like a subtraction on a pre-defined integer type and a user-defined function on a user-defined datatype. It is natural to embed a subtraction into the datapath defined by a hardware process. As a consequence, functions are inlined in the processes. This is done either explicitly by copying and adapting the function body into the position of the function call. Or it is done implicitly by preserving the function capsule and generating VHDL functions during synthesis. Either way, the synthesis semantics stay identical. The function operates directly on the arguments. Copy-in-copy-out operations are resolved and removed by hardware synthesis tools.

2.5.3 Polymorphic Objects

Since pointers are not synthesized with OSSS and C++ requires pointers for polymorphism, there is no way to express polymorphism in a synthesizable way apart from implementing it manually. To overcome this limitation, OSSS supports polymorphic value objects. These follow the isolated resource pool model as introduced in Section 2.3.

Using a datatype template named `osss_polymorphic<>`, *polymorphic value objects* are introduced. The following code example shows the modeling of these. On the left hand side OSSS source code is shown, and the right hand side shows an equivalent C replacement.

```
// OSSS code
class Base
{
  public:
  void set42(){}
};

class X : public Base {
  public:
  int my_field;
  void set42(){
    my_field = 42;
  }
  ...
};

class Y : public Base {
  ...
}
...

// in a code block:
osss_polymorphic< Base > pe;
pe = X();
pe.set42();
```

```
// C code
typedef struct {
  int my_field;
} X;

typedef enum { tag_X, tag_Y }
  type_tag;

typedef union {
  X variant_X;
  Y variant_Y;
} variant_union;

typedef struct {
  type_tag      tag;
  variant_union variant;
} poly_type;

...
void set42(poly_type this_obj)
{
  if (this_obj.tag == tag_X) {
    this_obj.variant
      .variant_X.my_field = 42;
    ...

// in a code block:
poly_type pe;
pe.tag = tag_X;
pe.variant_X.my_field = ...
...
set42(pe);
```

The `variant` field of the polymorphic element is re-interpreted according to the current value of the `tag` field. Each time the tag field is to be evaluated, a conditional statement (`if` or `switch`) is generated for this test. Depending on the outcome, a different variant is assumed. Inside each alternative block, the polyorphism is statically resolved. As a result, a hardware multiplexer structure is implemented. The synthesis of polymorphism based on the restricted resource pool model is mapped to the problem of synthesizing C `unions`, which in turn is only data re-interpretation.

In this thesis, instances of this polymorphic value type, are called *polymorphic value objects*. There are other objects of polymorphic nature introduced in later sections, which are not of a *value* type.

2.5.4 Shared Objects

As described before, variables that are accessed by multiple processes have an undefined behavior. The order in which a simulator invokes concurrent processes is not defined by the SystemC standard (for good reason) but may influence the model's simulation in such cases. For synthesis, this is even worse, because there is no way to provide a hardware implementation of this.

A designer facing such an issue might turn the variable into a signal. Signals have a well-defined behavior, since multiple reading processes are possible. Read operations always yield the value which was last written to the signal. Multiple writers are

forbidden, therefore the last written value is well-defined.

In case of multiple writers, an intermediate process can be written which manages the accesses to the variable. This is a undesirable situation, since the C/C++ model which is to be implemented in SystemC likely uses method invocations and function calls. Remember, that assignments to objects are invocations of the assignment operator in C++ and assignments to non-class variables can be treated as function calls. Manually transforming method invocations into a handshake mechanism with an extra management processes is undesirable.

Shared use of a variable is a way to express communication. SystemC signals do not support method based communication and hierarchical SystemC channels are not suitable for synthesis.

To overcome this issue, a new object kind was introduced in the OSSS project, called *shared object*. The concept is inspired by so-called *protected objects* from ADA. Shared objects are entity objects, which support access from multiple processes. A built-in arbitration component decides on-line, which request is to be served next. The requests are indicated by method invocations. Shared objects employ a user-defined class, which is used for both interfacing and implementation. Due to this user-defined class it is possible to invoke any kind of methods, not just read and write calls. This even includes operations on the shared variable, which may span multiple clock cycles. Having this at hand, true method based inter-process communication is possible. Second, since shared object methods may include operations, this entity object represents both a shared datapath and memory.

Note, that shared objects are *passive*. From a modeling point of view, they do not contain processes[4].

> **Definition 2** - ***Active:***
> *An active design element is able to initiate a communication without external stimulation. This requires having its own thread of control due to an embedded process and a state.*

The definition of *passive* is complementary to *active*. Examples for active elements that may be active are SystemC modules or hierarchical channels. Signals, ports or variables are examples for passive elements.

2.6 Dynamic Partially Reconfigurable Hardware

The process of replacing application logic is often called reconfiguration. This term is used in many different domains, for example software or analog hardware. In this thesis, we refer to digital hardware only.

Reconfigurable hardware can be classified as coarse-grained or fine-grained reconfigurable. This refers to the minimal complexity of elements, which are typically replaced. For coarse grained reconfigurable hardware this means units like digital filters, data compressors, or media codecs. Usually, the units are equivalent to some hundred logic gates or more. Fine grained reconfigurable hardware allows replacement of way smaller units, typically a minimal size equivalent to a few logic gates. Table 2.4 shows the different reconfiguration granularities.

One can build coarse grain reconfigurable systems using fine grain reconfigurable hardware. It would be less flexible, but more area-efficient to use custom coarse grain reconfigurable hardware. Since fine-grain reconfigurable hardware is more common and more general, this thesis does not cover coarse-grained custom hardware.

[4]This changes during synthesis. Shared objects become active but do not use the possibility to initiate a communication themselves.

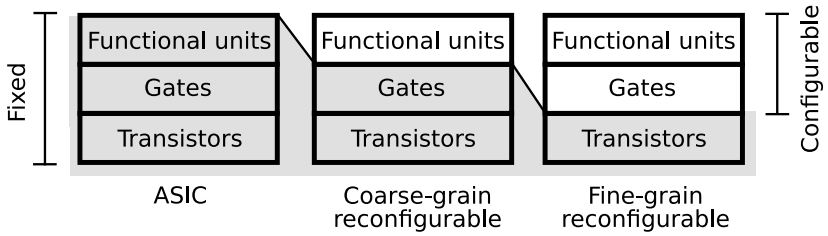

Figure 2.4: Different configuration granularities

The typical fine-grained reconfigurable hardware devices are FPGAs.

FPGA is an abbreviation for field-programmable gate array. As the name suggests, it is a matrix structure of logic gates. These gates are programmable "in the field", that means, after manufacturing. The term "configuration" is often used when referring to the programmability of these devices. This is usually done when referring to downloading a bitstream on the device while it is inside the final system (instead of using a dedicated external programming hardware to prepare the chip before mounting it).

Definition 3 - ***Configuration***:
A (FPGA-)configuration is a bitstream used to define the logic (e.g. look-up table definitions) and routing (use of wires and switches) for a FPGA area. The term configuration *can also refer to the process of loading the bitstream into the FPGA and putting it into effect.*

Definition 4 - ***Re-configuration***:
This refers to the process of a configuration overwriting a previous one.

Even the initial configuration after system power-on overwrites a previous random one. Therefore, in this thesis the simple term *configuration* is used for both initial configuration and re-configuration. However, not all FPGAs can be reconfigured. To explain this, it is helpful to classify FPGAs by the way the configuration data is stored. Table 2.2 gives an overview.

Especially the SRAM technology allows implementing flexible ways to perform configuration processes. We define the following special abilities:

Definition 5 - ***Partial reconfiguration***:
A partial reconfiguration is defined as the reconfiguration of a fraction of the FPGAs area.

Definition 6 - ***Dynamic reconfiguration***:
A dynamic reconfiguration is a configuration process while the system is in operation, this is sometimes called runtime reconfiguration (RTR), too.

Antifuse-based	*Antifuses* are electrical devices which have a high resistance after manufacturing. When the applied voltage exceeds a specified value, the resistance is lowered. This creates a conducting connection. Antifuse-based FPGAs are programmed by creating these connections. One disadvantage is that the process is not reversible. On the other hand, the configuration is kept permanently, even at power loss.
EPROM-based	These FPGAs use an erasable programmable read-only memory (EPROM) for configuration storage. EPROMs are erased by ultraviolet light. Since there is no light source on the device, they are programmable but not re-configurable.
EEPROM-based and Flash-based	These FPGAs use an erasable programmable read-only memory (EPROM) or *Flash* memory for configuration storage. In case of electrically erasable PROMs (EEPROMs), the FPGA has its own electrical circuitry to erase the data, so no source of light is needed. The Flash memory is also erasable. The most relevant difference is the granularity of data which can be erased. EEPROM based FPGAs are fine-grain erasable, while Flash FPGAs are block erasable.
SRAM-based	These are the most common FPGAs. The configuration data is stored in static RAM cells. These cells are fine-grain erasable. The area required by the configuration cells is small, compared to the other FPGA types. On the other hand, it is the only volatile technology among those listed here. This requires a configuration on each power-on.

Table 2.2: FPGA classification by configuration storage

Definition 7 - ***Self reconfiguration****:*
This is a reconfiguration process which is controlled from the FPGA itself. This includes the decision of which bitstream to use and when to reconfigure.

The combination of dynamic and partial reconfigurability is abbreviated as *DPR* in this text. In the following chapters, the term FPGA refers to dynamically partially reconfigurable devices, unless explictly stated otherwise. To ease describing FPGA properties in more detail, some further terminology is helpful:

Definition 8 - ***Configuration port****:*
The configuration port is the bit-level interface provided by the FPGA to read or write configuration data.

Definition 9 - ***Dynamic state****:*
The dynamic state is the state of sequential elements (registers) of the FPGA, which carry the application's data. This does not include the memories to store the logic configuration, e.g. look-up table definitions or routing information. Typically, the dynamic state is that kind of data which is initialized upon application reset.

Definition 10 - ***FPGA context***:
A FPGA context is the combination of configuration and dynamic state.

Most FPGAs hold exactly one context for the FPGA area, these are called *single-context*. To speed up reconfiguration, *multi-context* FPGA families were proposed (e.g. [101, 84, 135]). Only one context is active at any time but switching to another one is very fast (one to a few clock cycles).

Definition 11 - ***Dynamic and static design components***:
A design component (VHDL entity, SystemC module etc.), which is not designed to be possibly reconfigured is called a static design component. Complementary, if the component may be exchanged during runtime, it is called a dynamic design component.

2.6.1 Classification of DPR FPGAs

To understand the impact of DPR on the design process, it is helpful to clarify some characteristics of FPGAs doing DPR:

Granularity

Even fine-grained reconfigurable FPGAs have a minimum granularity in exchanging logic. In other words, even if one atomic element (a look-up table for example) is to be replaced, the configuration bitstream determines the programming of some neighboring elements, too. It is heavily technology dependent which cells are in a configuration neighborship. In Xilinx Virtex II[142] and Virtex II Pro families this was a frame of four elements width, ranging over the whole height of device. The successor families, Xilinx Virtex 4[141] and 5 [146], limited this to a height of 16 elements (in case of Virtex 4) and 20 elements (Virtex 5)[76].

Due to this, replacement of logic elements needs to be done carefully. If two reconfigurable areas share the same neighborship, reconfiguring one might affect the other. To overcome this, one might read the current logic configuration to a off-chip location (e.g. a host PC), modify the desired fraction only, and write it back onto the device. For example, this was possible for Xilinx Virtex II devices using the Xilinx JBits API[56].

Another problem arises if the configuration bitstreams for a FPGA architecture include the dynamic state for the affected area. Then the design needs to be built in a way that writing back an outdated dynamic state to the neighborship does not affect the circuits behavior.

In either case, reconfiguring the neighborship needs to be glitch-free, if the neighborship is occupied by non-interrupted logic (which is the intention of partial dynamic reconfiguration). This is an additional feature the FPGA architecture has to support. If these requirements are not met, the logic placement must leave the neighborship unused.

Reconfiguration Times

The time required to reconfigure is usually in the range of milliseconds. The exact duration is typically proportional to the size of the bitstream which in turn is closely related to the size of the reconfigurable area. Bitstream compression [33] may reduce the duration significantly. As long as fetching bitstream is done at a constant rate (not through a cache hierarchy, for example), the reconfiguration duration is deterministic, example bitstream sizes can be found in [76].

Loading a bitstream is in the same order of magnitude as loading a small software program on a current personal computer. While this is fast enough for a PC user, a problem for hardware development arises. A reconfigurable design instance (e.g. an entity in VHDL terminology) appears to be unreliable and unpredictable during the reconfiguration process which easily lasts some thousands of clock cycles. This is a far from an unnoticeable switch of configurations.

As a consequence, the static components potentially being in interaction with the dynamic component need to be aware of reconfiguration processes. They need to delay requests, buffer data etc. to compensate for the reconfiguration.

Configuration Infrastructure

ASICs and FPGAs both allow a high degree of parallelism compared to software implementations. This is exploited when designing small components and validating their functionality in isolated simulations and tests. One paradigm is that each component may compute isolated from others. Since there are no shared bottlenecks (unlike for example in von-Neumann architectures), the components do not influence each other.

When it comes to DPR, a bottleneck is being introduced: The reconfiguration infrastructure. First, there is usually one configuration port implemented on each device[5]. The same applies to the source of configuration bitstreams. Flash memories, EPROMs and other memories cannot be accessed in arbitrary parallelism and with any bandwidth. Again, this imposes a restriction: There are no truly parallel reconfiguration processes. One may introduce pseudo-parallelism by using interleaved configuration, if the FPGA architecture does permit it but this extends reconfiguration duration.

As a consequence, there must be some arbitration of the reconfiguration port and bitstream source. This can be done by avoiding conflicts using a specially crafted and pre-calculated schedule (*offline schedule*) at the cost of design flexibility. If the reconfiguration schedule cannot be determined statically, run-time scheduling (*online scheduling*) is required. This needs to be implemented in the static part of the design as well.

This mutual influencing of sub-designs requires a careful analysis, design and validation of the system's behavior.

Opportunities and Risks

Replacing idle subcircuitry and re-using the occupied area bears potential to reduce the size of circuits. This may allow using a smaller FPGA device at the cost of a larger bitstream storage.

DPR may yield several benefits:

- Cost savings: The bitstream storage may be cheaper than using a larger FPGA device, this may also reduce the required size on the printed circuit board.
- Power savings: A whole idle non-volatile bitstream source may be put into power saving modes more easily than a fraction of a FPGA area. A reduced FPGA area reduces static power consumption compared to a larger device of the same family.
- Updates in the field: A suitable bitstream source may be updated with optimized or bug-fixed implementations for algorithms. This is not possible with ASICs at all, which in turn is the only alternative implementation technology providing at least the same computational performance in comparison to the FPGAs.

[5] Some FPGAs have two, e.g. Xilinx Virtex 4, but they cannot be used concurrently.

On the other hand, DPR may bring the following disadvantages:

- Reduced family set: A designer may choose only among SRAM based FPGA families and needs to ensure they are capable of DPR. This may be prohibitive where the platform is already fixed or operating conditions suggest the use of other technologies (e.g. antifuse-based FPGAs in radiation-intensive environments such as satellites).

- Increased energy consumption: During reconfiguration, FPGAs often consume more power than in normal operation mode. Additionally, calculation results may be delayed due to reconfiguration, leading to longer runtimes (and less time for power-saving modes). This may outweigh the power savings due to the reduced area.

As a consequence, the decision whether to use DPR cannot be made without carefully looking at the application, the operating conditions, device prices, power supply etc.

2.6.2 FPGA Architectures in the Market

Over the last years, several vendors introduced FPGAs with abilities for reconfiguration. Xilinx Inc. offered the most well-known FPGA families in this respect. Several research groups used the Xilinx XC62xx device, which offered a very fine-grain control for circuit modification. The XC62xx is now discontinued as well as its successor, the first "Virtex" device family.

Three successor families, Virtex II, 4 and 5 are on the market as of today. None of these allows control that is as fine-grained as it was provided the XC6200 family. Virtex II devices[142], for example are configured in frames, which span the whole height of the device. This imposes limitations on the layout of a reconfigurable circuit. Having multiple reconfigurable areas quickly leads to a situation where communication needs to be done through reconfigurable areas. While this was possible, it was difficult to achieve using the earlier tool flows[6]. An additional problem was the logical height of the device differing with the amount of on-chip logic elements. A third obstacle was the placement of I/O cells. Since they were located at the borders of the chip, placing reconfigurable areas at these borders would also occupy all I/O pins located at the left or right device side.

There has been academic work to provide prototyping platforms supporting partial reconfiguration that use a two-chip approach [23]. The additional chip provides a crossbar and additional I/O pins to circumvent these problems of Virtex II devices.

With the advent of the Virtex 4 and 5 families [141, 146], the situation was improved. First, reconfiguration frame height was fixed to a height of 16 logic elements (Virtex 4) or 20 logic elements (Virtex 5) and the device height was composed of several frames. Now, communication through a reconfigurable area was no longer necessary. Second, the I/O ring at the device borders was replaced by distributed I/O-pins over the chip area.

Xilinx' Virtex devices offer self-reconfiguration abilities since the reconfiguration port can be accessed from within the chip logic. The low-cost Spartan-3 series [145] does not offer such an internal configuration port. However, several publications [30, 62, 19] have shown that self-reconfiguration is possible with a suitable board design.

Atmel introduced an reconfigurable FPGA family named AT40k[17]. Dynamic partial reconfiguration may be done using an external AVR microcontroller. An AT40K FPGA, an AVR microcontroller and an integrated 36 kilobyte SRAM along with other

[6] In earlier flows, no static design parts were allowed to reside inside a reconfigurable region. Routing a signal through the region was to be implemented individually (and potentially different) by each reconfigurable module. During reconfiguration, these signals were interrupted, if the route was changed.

Vendor	FPGA family	Reconfigurable	Dynamically reconfigurable	Partially reconfigurable
Xilinx	Virtex 2	yes	yes	yes
	Virtex 4	yes	yes	yes
	Virtex 5	yes	yes	yes
	Spartan 3	yes	yes	yes
Altera	Cyclone III	yes	no	no
Atmel	AT40K/AT94K	yes	yes	yes
Lattice	LatticeXP	yes	no	no

Table 2.3: Comparison of mainstream FPGA families

smaller components are offered under the name AT94K [18]. Since the AT94K consisted of several components typically found in a System-on-Chip design Atmel called it a "Field Programmable System Level Integrated Circuit" (FPSLIC). These Atmel devices have been used in the Figaro[85] design flow developed in the RECONF2 project.

Lattice offers LatticeXP[65] FPGAs, which store their configuration in both a flash memory and a SRAM memory. The flash memory is used to permanently store the configuration, making it non-volatile. This configuration data is copied to distributed SRAM to apply the configuration and enable it. The company advertises "Transparent Field Reconfiguration" (abbreviated TransFR oder TFR), which is a four-phase process to update the configuration of a device. First, the flash memory is updated with a new configuration. Then the I/O pins are locked to avoid glitches during activation of a new configuration. Third, the flash memory is copied to the SRAM and finally the normal operation mode is entered again. Although this is not partial configuration and needs to be controlled externally by a processor, it can still be considered dynamic configuration. This technique is usable for updates in the field, e.g. to extend an application or correct design flaws.

Altera offers devices with an auto-configuration mechanism but no partial configuration. For example, the Cyclone III[13] has an update ability, which makes FPGA stop executing and installs a complete configuration. This is comparable to Lattice's TransFR feature.

Table 2.3 gives an overview about the reconfiguration abilities of some current mainstream FPGA families.

2.6.3 Industrial Tool Flow

To the authors best knowledge, Xilinx is the only company today offering tools flow for partial reconfiguration that supports self-reconfiguration without involving a processor core. A similar flow could be implemented for Atmel's AT40K family, but the current flow requires external configuration control by a microprocessor.

Xilinx' "Difference-Based Partial Reconfiguration Flow"[45] requires the designer to implement a static design including bitstream generation. There were small logic changes possible using tools like Xilinx FPGA-Editor. Finally, differential bitstreams covering these changes were generated. This flow is feasible, if small changes are to be made to a configuration and if small differential configuration bitstreams are desired.

The "Module-Based" design flow is more sophisticated ([81, 22], also described in [45] up to rev. 1.2). Modules are FPGA regions that communicate with their environment through so-called *bus-macros*. Bus macros are logic elements programmed to act as "waypoints" for signal routing. All signals crossing the border of a module must be routed through these macros. For that purpose, bus-macros are located at the physical borders of a module. By connecting internal and external signals to the macros, the module can be replaced without breaking routing, as long as the bus-macros stay in

the same position. They would be unnecessary, if the router could be forced to implement a signal to use a specific wire inside the FPGA, which crosses the module's border. The different modules are synthesized independently and then combined and floorplanned. Although this flow is more powerful than the difference-based flow, it was still very difficult to use it due to technical reasons. One example is that the router could not be forced to strictly obey the module's area limits when routing signals. Many workarounds for these issues were found over the years, yet it still remaines a difficult process.

The current flow is called "Early Access Partial Reconfiguration" (EAPR) flow[143]. It is similar to the module-based flow but is easier to use. The flow distinguishes between partial reconfigurable regions (PRR) and partial reconfigurable modules (PRM). PRMs are the different alternatives to reside inside a PRR during runtime. In a floorplanning step, the PRRs are defined. Then, the static design part is synthesized and placed. After the static part is implemented, the remaining resources inside the PRRs are used to implement the PRMs. A modified router ensures that PRM-internal signals are not placed outside the PRR. Finally, an initial full configuration bitstream is generated as well as individual partial bitstreams for PRMs.

With the introduction of the Virtex 4 and 5 families, as well as the EAPR flow, Xilinx greatly reduced the design effort in the backend flow to implement a partially reconfigurable design. However, the task of providing a design, which can be separated into reconfigurable modules, and manages reconfiguration itself is still up to the designer.

2.7 Chapter Summary

This chapter introduced the foundational basis for the proposed approach. It covered relevant features of C++ with respect to the modeling elements to be introduced in chapter 4. It then shortly introduced SystemC and OSSS for modeling hardware. Finally, partial reconfiguration was discussed along with important architectures and tool-flows.

Chapter 3

Related Work

There has been intensive research in the field of dynamic partial reconfiguration (DPR) for about 15 years now. The potential of this new technique for various types of applications was investigated and many case studies were made. New architectures were built to exploit the special abilities of reconfigurable hardware. Among these were many coarse-grained architectures, e.g. MorphoSys[120], MaRS[127], RoSA[95], or Totem RaPiD[31]. An overview on coarse grain reconfigurable architectures can be found in [57].

Others used reconfigurable components as co-processors or flexible datapath extensions for general purpose processors, examples are NEC DRP[14], works by Vuletić et al.[134], the MuCCRA architecture[132] or DISC[140]. Adres[79] is a combination of a coarse-grained architecture and a microprocessor.

Further architecture, framework and methodology overviews are presented in [117, 24, 32, 130, 49]

The invention of these new architectures raised the question of appropriate design flows to hide the additional complexity introduced by reconfigurability. The same question even arises for use of DPR in a purely hardware context on mass-produced fine-grain reconfigurable FPGAs.

Even for these commercially available FPGAs, a lot of approaches were published. In this chapter, an overview of the approaches being closest to the topic of this thesis will be presented. They generally have the following properties:

- The architecture target is a fine-grain reconfigurable FPGA capable of DPR and using that feature.
- The approach supports at least simulation or synthesis of designs, not just abstract modeling.
- The input specification is expected in terms of a textual hardware description language.

Besides these, there are many approaches which sacrifice timing-accurate simulation of the design entry model in favor of using established hardware description languages, for example for the DYNASTY [133] framework, and works in the RECONF2 project[116].

One can argue that the DCS framework (see VHDL-based approaches below) does belong to this category, too. But since it adds a dedicated reconfiguration information file as part of the input specification, it will be covered in more detail below.

3.1 Categorization

To allow a systematic comparison, the related work is characterized using a scheme. Modeling, simulation and synthesis aspects are described separately.

Modeling Aspects

Level of Abstraction. This refers to the kind of description used for design entry. Possible alternatives are:

1. *Application*: An algorithmic or behavioral description, e.g. a C/C++ specification or abstract SystemC.
2. *TLM*: The model's components interact using transactions (typically `read()` and `write()` calls on busses or abstract channels).
3. *Register transfer*: An HDL specification, e.g. Verilog, VHDL or RT-level SystemC model.
4. *Other*: A graphical entry (e.g. diagrams or charts), a graph specification, or other abstract specification.

Expression of DPR requests. How is demand for reconfiguration expressed? Possible alternatives are:

1. *Fixed schedule*: The schedule of reconfigurations is known at design time.
2. *Runtime condition:* Boolean expressions evaluated at runtime indicate reconfiguration conditions.
3. *Explicit*: The schedule is calculated or updated at runtime. The application requests DPR actions by explicit statements.
4. *Implicit*: The schedule is calculated or updated at runtime. The application code contains usage statements for certain logical objects, bus members or similar abstract addressing. This implicitly expresses the needs for DPR reconfigurations or state copying actions if the run-time situation does not match the demand.

Dynamic state persistency. What happens to state during switch actions? Possible alternatives are:

1. *None*: State cannot be preserved and restored.
2. *Zero-time*: Re-enabling a previous configuration includes automatic recovering of the previous state. This process is assumed to happen instantaneously.
3. *Manual*: Saving and loading needs to be implemented manually but is then reflected with accurate timing.
4. *Automatic*: Saving and loading done automatically.

Reconfiguration control. What type of component performing bitstream upload? Possible alternatives are:

1. *Does not apply*: Not mentioned or not applicable to the approach (e.g. due to being a simulation-only approach).
2. *Hardware*: A hardware logic does perform the upload. So there is no processor required.
3. *Software*: A processor is required.

Multiple reconfigurable areas. Are multiple reconfigurable areas supported? Possible alternatives are:

1. *No/Not shown*: Single area only, no conflicts possible. This does not indicate that there is no way to extend the approach to support multiple reconfigurable areas. It rather says that there was no support for this feature shown in literature.
2. *Offline scheduled*: Reconfigurations in fixed sequence
3. *Online scheduled*: Reconfigurations scheduled at runtime

Shared use of reconfigurable areas. Areas used by multiple HDL processes or bus masters? Possible alternatives are:

1. *No/Not shown*: Single process using a reconfigurable area, single bus master or other restricted control.

2. *Offline scheduled*: Yes, accesses fixed at compile time
3. *Online scheduled*: Yes, accesses scheduled at runtime

Simulation

First, it is being answered, whether there is a simulation ability published or not.

Accuracy. How close is the simulated behavior to the final circuit? Possible alternatives are:

1. *Weak*: Maybe somehow timed, but not close enough to the timing of the resulting circuit. Benchmarking is not possible or yields vague numbers.
2. *Coarse timed*: Few clock cycles difference at most.
3. *Cycle accurate*: Clock cycles match exactly.

Reconfiguration overhead. Are reconfiguration times reflected in simulation? Possible alternatives are:

1. *Omitted*: No, switches happen instantaneous.
2. *Included*: Yes, reconfiguration times are reflected.

Persistency overhead. Is state preservation or restauration time included? Possible alternatives are:

1. *Does not apply*: There is no state preservation modelled.
2. *Omitted*: No, states saved and loaded instantaneously.
3. *Included*: Yes, state saving and loading times are visible.

Scheduling overhead. In case of online reconfiguration scheduling, is the overhead reflected in the simulation? Possible alternatives are:

1. *Does not apply*: There is no reconfiguration conflict possible (e.g. due to a single reconfigurable area only).
2. *Omitted*: Scheduling happens in zero-time.
3. *Included*: Scheduling cycles are modelled.

Synthesis

First, it is being answered, whether there is a synthesis flow published or not, even if it would have to be done manually.

Tool support. How is the model transformed to HDL code that can be processed by FPGA vendor tools? Possible alternatives are:

1. *None*: Manual re-implementation required.
2. *Limited*: Some steps have tool support.
3. *Full*: All necessary steps are done by tools.

3.2 Related Approaches

3.2.1 SystemC Based Approaches

This section presents approaches that are based on SystemC and therefore have a strong relationship to OSSS+R.

Dynamic Module Library

Modeling	Level of abstraction	Application or TLM
	Expression of DPR requests	Runtime condition
	Dynamic state persistency	None
	Reconfiguration control	Does not apply
	Multiple reconfigurable areas	No/Not shown
	Shared use of reconfigurable areas	No/Not shown
Simulation (yes)	Accuracy	Weak
	Reconfiguration overhead	Omitted
	Persistency overhead	Does not apply
	Scheduling overhead	Does not apply
Synthesis (no)	Tool support	None

The *Dynamic Module Library*(DML)[16, 63] is a framework intended to allow modeling of reconfigurable systems.

The approach supports modules to be exchanged during runtime. The technical basis is thread spawning, which was introduced with SystemC 2.1v1 and is now a part of the IEEE 1666-2005 standard SystemC 2.2.0. Thread spawning is intended for use in testbenches, so the library features are tailored to this purpose. It is difficult to dynamically add and remove whole modules, since the signal and port binding is fixed after the elaboration phase. To overcome this problem, DML provides dynamic port containers, e.g. `dc_port` that allow connecting and releasing signal and port bindings at runtime.

Each dynamic module is equipped with a user code process and a spawn control process handling deletion requests. The spawn process monitors deletion conditions and notifies the user process, if required. User processes are required to be of thread type and therefore contain a `while(true)` loop. This loop is exited if notified by the spawn control. This setup allows modeling of processes to be exchanged. They can be (re-)active and perform complex communication protocols.

Module switches discard the previous dynamic states. There is no hook provided to model a state preservation prior to module destruction. If required, this needs to be unfolded into the application logic. There is no support for multiple reconfigurable areas inside one system. The resulting conflicts are not reflected in the proposed model. Since the description is on register transfer level, there is also no support for shared use of dynamic modules. If necessary, this also needs to be unfolded into the application logic. Finally, there is no synthesis demonstrated. This also affects simulation accuracy, since reconfiguration times are not known and not simulated.

ReChannel

Modeling	Level of abstraction	TLM or register transfer
	Expression of DPR requests	Explicit
	Dynamic state persistency	Manual
	Reconfiguration control	Hardware or software
	Multiple reconfigurable areas	No/Not shown
	Shared use of reconfigurable areas	No/Not shown
Simulation (yes)	Accuracy	Cycle accurate
	Reconfiguration overhead	Included
	Persistency overhead	Included
	Scheduling overhead	Does not apply
Synthesis (yes)	Tool support	None

The ReChannel approach [103, 104] is a modeling library that guides a designer when implementing reconfigurable systems in SystemC. The intent of this approach is not to provide a maximum of tool support. Instead, it provides a technical and methodological guide that aids the designer from a very abstract level down to implementation level without enforcing restrictions.

ReChannel allows exchanging SystemC modules at runtime. There is no restriction for these modules like single-process only or certain process types. Reconfigurable modules need to be derived from a `rc_module` class provided by the library. A reconfiguration controller of type `rc_control` is to be instantiated and instructed about the reconfigurable modules. The application logic requests reconfigurations by explicitly invoking methods on the controller (`load`, `activate`). If provided by the designer, loading, activation and removal times are taken into account during simulation when performing these commands. Communication with the reconfigurable modules is done via *portals*, which are specialized channels performing event forwarding. From a functional point of view, portals act as multiplexers, yet they are realized differently. A performance study revealed that the simulation is likely to be faster than a hand-crafted multiplexer solution. In addition, portals don't even induce delta-delays during the simulation, which would be unavoidable by a multiplexer solution.

Multiple reconfigurable areas and possible reconfiguration conflicts are not covered in literature. The same applies to shared use of the reconfigurable area but this is very consistent with the overall philosophy of the approach. Shared use (by multiple processes) needs to be unfolded into the application logic. The same applies to the saving and restauration of a reconfigurable module's dynamic state. However, it is possible to reduce this step to elapse a certain amount of time for simulation to save early design effort. The real implementation can be added later on.

ReChannel aims at a widely unrestricted design space, which makes it hard to assist the designer with tools. All refinement steps are guided by the library, yet being done manually. This applies in particular to the reconfiguration controller, which needs to be implemented manually.

UF Campina Grande

Modeling	Level of abstraction	TLM or register transfer
	Expression of DPR requests	Fixed schedule or runtime condition
	Dynamic state persistency	Zero-time
	Reconfiguration control	Hardware
	Multiple reconfigurable areas	No/Not shown
	Shared use of reconfigurable areas	No/Not shown
Simulation (yes)	Accuracy	Cycle accurate
	Reconfiguration overhead	Included
	Persistency overhead	Omitted
	Scheduling overhead	Does not apply
Synthesis (no)	Tool support	None

In [27, 26] a modified SystemC kernel is used for simulation of reconfigurable systems. The approach is intended to allow early exploration of design alternatives.

The kernel was extended by routines allowing to enable and disable modules. `SC_METHOD`s inside disabled modules are never invoked until the module gets enabled again. This does not influence thread-type processes, allowing register transfer specifications with explicit state machines for use inside reconfigurable modules only. The reconfiguration process is under the control of a dedicated hardware module, called `ConfigurationManager`. It follows a fixed sequence in literature but it should be easy to use a more flexible implementation. The approach allows specifying reconfiguration times which are then used for simulation. Dynamic state preservation times are not specified. One could simply include state loading times in the reconfiguration time for a module. State saving times depending on the kind of module to be disabled is not supported. Conflict resolution for concurrent reconfiguration requests in case of multiple reconfigurable modules was not demonstrated yet. Use of a single reconfigurable module by multiple external processes needs to be unfolded into the application logic. The simulation is timing accurate, as long as no state preservation is involved. There is no synthesis concept demonstrated, because the approach is intended for early exploration whether DPR is applicable or not.

SystemC DRCF

Modeling	Level of abstraction	TLM
	Expression of DPR requests	Runtime condition
	Dynamic state persistency	Zero-time
	Reconfiguration control	Does not apply
	Multiple reconfigurable areas	No/Not shown
	Shared use of reconfigurable areas	No/Not shown
Simulation (yes)	Accuracy	Weak
	Reconfiguration overhead	Omitted
	Persistency overhead	Omitted
	Scheduling overhead	Does not apply
Synthesis (no)	Tool support	None

In the ADRIATIC project a SystemC based flow for design space exploration, simulation and synthesis of reconfigurable systems is proposed[92, 129, 78]. It aims at being technology-vendor independent and is meant to integrate well into commercially available design tools.

Modules to be present mutually exclusive are to be modelled as bus slaves, being identified by an address range. To introduce reconfigurability, groups of them are wrapped in other modules, called dynamically reconfigurable fabric (DRCF). From an

outside view, a DCRF behaves like a single bus slave. Internally, it switches to the appropriate wrapped module according to the address given on the bus. DCRF forwards bus operations like `read(...)` and `write(...)` internally. The introduction of DRCFs is supported by a tool developed in the project. The description is done at the transaction level, since the communication partners are bus masters and slaves. Modules being disabled just loose their connection to the outside, this preserves their internal state. However, since no delays are specified for reconfiguration or persistency, the simulated timing does not match the final circuit very well. This approach requires a manual source-code transformation to obtain a synthesizable model.

The ADRIATIC project also used a specification language called OCAPI-xl, see Section 3.2.2.

SyCERS

Modeling	Level of abstraction	Application or TLM
	Expression of DPR requests	Runtime condition
	Dynamic state persistency	None
	Reconfiguration control	Hardware
	Multiple reconfigurable areas	No/Not shown
	Shared use of reconfigurable areas	No/Not shown
Simulation (yes)	Accuracy	Coarse timed
	Reconfiguration overhead	Included
	Persistency overhead	Does not apply
	Scheduling overhead	Does not apply
Synthesis (no)	Tool support	None

SyCERS [15] is a framework that allows modeling of runtime reconfiguration. It is intended to help designers to explore and validate reconfigurable solutions. In this approach, the functionality to be replaced is represented by functions inside modules. These functions are called from the body of `SC_THREAD`s and `SC_METHOD`s. By using function pointers to change the function at simulation time the dynamic behavior is achieved. Since, functions are switched between calls, there is no local state inside the function to handle (neither control flow state nor local variables). This is accurate modeling, if the dynamic state (which are the module data members and signals) of the module containing the functions to be switched is realized outside the reconfigurable area in static hardware implementation.

Despite that it is not explicitly described how reconfiguration times are taken into account, the article [15] (see Section 4.3.2, last paragraph) suggests that there is some way to specify them. The sequence of required configurations is directed via bus addressing by the application connected to the reconfigurable part. Therefore, no access scheduling is performed inside the reconfiguration controller. The authors do not demonstrate the use of multiple reconfigurable modules, eliminating the need for a reconfiguration ressource scheduling.

Simulation is done by mapping to the Caronte[42, 115] architecture, containing a microprocessor and a bus system to access the reconfigurable modules. There is no support for synthesis shown.

SystemC'mantic

Modeling	Level of abstraction	Other (KPN in SystemC)
	Expression of DPR requests	Fixed schedule or runtime condition
	Dynamic state persistency	None
	Reconfiguration control	Does not apply
	Multiple reconfigurable areas	No/Not shown
	Shared use of reconfigurable areas	No/Not shown
Simulation (yes)	Accuracy	Coarse
	Reconfiguration overhead	Included
	Persistency overhead	Does not apply
	Scheduling overhead	Does not apply
Synthesis (no)	Tool support	None

SystemC'mantic [64] uses SystemC to model an application defined as a Kahn Process Network. The approach is intended to shorten the design time from a functional description down to a TLM or register transfer level model.

To allow runtime creation and destruction of modules, the SystemC kernel was modified. Additionally, module connections may be modified, too. This can be used to vary the level of detail of a module during runtime, e.g. replacing an abstract model with a finer one during runtime. It is also possible to model a reconfigurable system this way. The authors have added *delay support* to the kernel, which indicates that at least reconfiguration times are modeled. The authors of [64] demonstrate the mapping of a 802.16a modem to software. It is not clear how the flow for reconfigurable hardware would work.

The demonstrated example shows a fixed schedule for reconfigurations although it seems to be easily extendable to monitor reconfiguration conditions.

PERFECTO

Modeling	Level of abstraction	Application or TLM
	Expression of DPR requests	Explicit
	Dynamic state persistency	None
	Reconfiguration control	Does not apply
	Multiple reconfigurable areas	No
	Shared use of reconfigurable areas	Online scheduled
Simulation (yes)	Accuracy	Cycle accurate
	Reconfiguration overhead	Included
	Persistency overhead	Does not apply
	Scheduling overhead	Included
Synthesis (no)	Tool support	None

Perfecto[59] was designed as a system design performance evaluation framework to support early design space exploration. It imposes a basic system architecture of a processor, memory, an arbitration component with a certain strategy, a function ROM, and a reconfigurable logic component.

To model partial reconfiguration, the dynamic hardware logic has to be placed inside the function ROM, described as C++ methods. In addition to their required arguments for the application, the functions are being passed configuration times, execution times, task numbers etc. Consequently, the method bodies contain application logic, wait statements (for configuration and execution delays) and also timing calculations. There is only little separation between runtime reconfiguration aspects and implemented logic.

The reconfigurable parts are modeled as methods operating on a stateless ROM, therefore no dynamic state persistency is required. Reconfiguration control is not implemented, as there is no synthesis path to hardware.

The framework assumes a slice-based reconfigurable area, where each reconfigurable logic may occupy parts of it. Placement is taken into account and logic re-use is also supported. This makes shared use of the reconfigurable area possible. This flexibility partly compensates for having one reconfigurable area only. Perfecto has a default scheduling mechanism implemented, which controls bus accesses and thereby reconfiguration sequences. In conjunction with reconfiguration times and the absence of dynamic states this should give the designer a cycle accurate simulation framework. The difficulty is rather to obtain the correct cycle count for execution times, reconfiguration times (inclusive protocol overhead) etc.

While the approach suffers from weak separation between application logic and implementation aspects, its strength is the ability to automatically generate reports. This disburdens the designer to analysing simulation traces to obtain figures like bus conflict statistics.

ITI Lübeck

Modeling	Level of abstraction	Application level or transaction level
	Expression of DPR requests	Runtime condition
	Dynamic state persistency	Manual
	Reconfiguration control	Software
	Multiple reconfigurable areas	Online scheduled
	Shared use of reconfigurable areas	Online scheduled
Simulation (yes)	Accuracy	Cycle accurate
	Reconfiguration overhead	Included
	Persistency overhead	Does not apply
	Scheduling overhead	Included
Synthesis (yes)	Tool support	Limited

The University of Lübeck developed an approach[11, 12, 97] that allows prototyping reconfigurable applications based on their CoNoChi network-on-chip architecture. It is comparable to the SyCERS approach presented in this chapter since both approaches use function pointers. The pointers are being called from the bodies of SystemC processes and methods. Switches in the function being pointed at resembles a reconfiguration. A reconfiguration management unit follows a state machine where each state represents a possible configuration. The transitions of the state are reconfigurations that have an associated cost. The management unit tries to limit the costs of its decision, e.g. to avoid configuring large fractions of the reconfigurable area. This optimization is dependent on the current system load. The reconfiguration control was demonstrated as a software implementation on an embedded processor. Nonetheless it would be possible to use a dedicated hardware unit instead.

The underlying architecture is a flexible *network on chip*. The reconfigurable area is logically divided into a set of tiles. A functionality can be associated to a set of tiles at runtime. Therefore, shared use of reconfigurable areas is possible. The dynamic state of such an area is not automatically preserved in case of reconfigurations. It is the application's task to implement persistency, if required. This could be difficult to achieve, since there is no synchronization between the reconfigurable modules and the reconfiguration management. The authors demonstrated stateless applications for their approach. These naturally do not have an issue here.

The authors presented a simulation framework, that allows back-annotating reconfiguration times. In principle, this makes the simulation cycle-accurate but time-consuming. The execution times for online-scheduling of reconfigurations are taken

into account. The scheduler is analyzed with respect to its execution times before simulation times and these runtimes are then annotated to the model.

A tool flow was not published yet, but publications on this topic are planned. Although there won't be a comprehensive tool for all tasks, support for individual steps can be expected.

3.2.2 Other C Language Style Approaches

OCAPI-xl

Modeling	Level of abstraction	Application
	Expression of DPR requests	Implicit
	Dynamic state persistency	Zero-time
	Reconfiguration control	Hardware or Software
	Multiple reconfigurable areas	Online scheduled
	Shared use of reconfigurable areas	Online scheduled
Simulation (yes)	Accuracy	Weak
	Reconfiguration overhead	Omitted
	Persistency overhead	Omitted
	Scheduling overhead	Included
Synthesis (yes)	Tool support	Limited

OCAPI-xl [129, 78, 99] was extended [106] to support run-time reconfigurable systems. OCAPI-xl processes are communicating via a message passing scheme. Reconfigurable processes can reach a state in which they are not being configured to hardware. Then, they appear to be blocked from an outside view. Sending messages to these processes will cause them to be configured to the hardware again. A static process called *HardWare Scheduler* controls the reconfiguration. Because reconfiguration is transparent in the OCAPI-xl model for the process being reconfigurable, the state is preserved. The authors in [106] do show how reconfiguration times or state preservation are modeled. It is possible to have overlapping ressources for reconfigurable processes. This leads to a shared use of a reconfigurable area which is dynamically scheduled. The authors explicitly point out that to ressources overlap in the given example. Therefore one can assume that this need not be the case. Consequently, multiple reconfigurable areas should be possible and are then dynamically scheduled. There is tool support to translate the model into VHDL with the exception of the hardware scheduler.

There are works using both OCAPI-xl and SystemC, see Section 3.2.1.

VPRS

Modeling	Level of abstraction	Register transfer
	Expression of DPR requests	Explicit
	Dynamic state persistency	None
	Reconfiguration control	Does not apply
	Multiple reconfigurable areas	No/Not shown
	Shared use of reconfigurable areas	No/Not shown
Simulation (yes)	Accuracy	Cycle accurate
	Reconfiguration overhead	Included
	Persistency overhead	Does not apply
	Scheduling overhead	Does not apply
Synthesis (no)	Tool support	None

Virtual Prototype for Reconfigurable Systems (VPRS)[100] is a C++ library providing hardware description abilities. The goal of this approach is to provide a fast simulation framework for early investigation of dynamic partial reconfiguration.

The authors treated extending the SystemC kernel as a greater effort than implementing the library from scratch. In VPRS reconfiguration is done on a module level. First, a *Slot* module is derived from a predefined class named `VPRSReconModule`. The slot acts as an interface description to the reconfigurable instance. Then, the implementation is further derived from the slot.

Reconfigurations are issues explictly using a `reconf()` template method on the modules. Reconfiguration times are explicitly added to the call in terms of clock cycles. This makes the simulation cycle accurate, as long as a real cycle count is at hand. Because there is no synthesis path, it is difficult to obtain such values.

Displaced modules are said to be destructed, therefore no dynamic state is saved. Since the library is intended to simulate a virtual prototype no reconfiguration control concept was included.

Tiles Methodology

Modeling	Level of abstraction	Structural register transfer
	Expression of DPR requests	Not shown
	Dynamic state persistency	None
	Reconfiguration control	Not shown
	Multiple reconfigurable areas	Not shown
	Shared use of reconfigurable areas	Not shown
Simulation (no)	Accuracy	Not shown
	Reconfiguration overhead	Not shown
	Persistency overhead	Does not apply
	Scheduling overhead	Does not apply
Synthesis (no)	Tool support	None

Lee and Milne [67, 68] proposed a design flow based on tiles, which are to be described using common HDLs. The authors observed that dynamic creation and destruction of objects is heavily used in software programming but lacks acceptance in the hardware domain. They propose a concept and a nomenclature to aid improving the situation in the design of programmable hardware.

The tiles can be *abstract* or *concrete*. Abstract tiles define an interface to a tile, whereas concrete tiles implement this interface. Tiles can be *composite* or *primitive*. As the name suggests, composite tiles are an aggregation of tiles while primitive ones are the leafs of such aggregations. The authors propose a state model to manage the allocation, modification and destruction of such tiles at runtime.

Even though neither tool-support for simulation nor synthesis abilities is shown, this approach has similarities to the approach proposed in this thesis and is listed here for that reason. The concept of abstract and concrete tiles resembles interface and implementation classes as used in object oriented modeling. The use of Liskov's substitution principle[72] makes the relation between interface and implementation comparable to inheritance relations.

The authors do not specify how to control DPR within the application. It is likely to be very application dependent and platform specific. Although dynamic state persistency is not shown, it could be well implemented manually. The same applies to the use of multiple reconfigurable areas and the shared area use.

RT-C

Modeling	Level of abstraction	Application or register transfer
	Expression of DPR requests	Implicit
	Dynamic state persistency	None
	Reconfiguration control	Software
	Multiple reconfigurable areas	Online scheduled
	Shared use of reconfigurable areas	Online scheduled
Simulation (no)	Accuracy	None
	Reconfiguration overhead	Does not apply
	Persistency overhead	Does not apply
	Scheduling overhead	Does not apply
Synthesis (yes)	Tool support	Limited

T.K. Lee at al.[69] used RT-C to describe a reconfigurable system. RT-C is a C-style language with focus on hardware-software communication and description of reconfigurable components. The reconfigurable elements are tasks, which can be invoked from outside. Reconfigurable tasks are grouped in structs, arrays or unions. The grouping determines the replacement, e.g. members of a union are mutually exclusive. Additionally, dynamically parameterizable variables are allowed. Modifications of them cause a runtime-reconfiguration.

The description of reconfiguration is implicit. The designer specifies, which task is needed at a position in control flow. Whether a reconfiguration is required to service this demand, is not explicitly specified. A run-time control system performs and tracks reconfigurations.

Tasks are not allowed to access global variables. They operate on their input arguments only, therefore there is no state to be preserved at task reconfigurations. Although the use of multiple reconfigurable areas was not mentioned, it should be no problem with this approach. Shared use of single reconfigurable areas is provided, since tasks in a task struct, union or array may be invoked by different callers.

There is no specification of reconfiguration or scheduling times given and the flow does not support simulation.

An example model was transformed into Handel-C and RTPebble, with tool assistance and some manual work. It was then implemented on a Celoxica RC1000-PP board. The reconfiguration control was done in software using Xilinx JBits API[56].

JHDL

Modeling	Level of abstraction	Structural register transfer
	Expression of DPR requests	Explicit
	Dynamic state persistency	None or manual
	Reconfiguration control	Software
	Multiple reconfigurable areas	Online scheduled
	Shared use of reconfigurable areas	No/Not shown
Simulation (yes)	Accuracy	Weak
	Reconfiguration overhead	Omitted
	Persistency overhead	Included (if done manually)
	Scheduling overhead	Does not apply
Synthesis (yes)	Tool support	Limited or full

JHDL[20, 60] is a structural HDL that was extended to cover reconfigurable circuits. The intention for its extension to support reconfiguration was to support describing both reconfigurable circuits and an external reconfiguration control for these. JHDL

uses language elements of Java to denote reconfigurations of FPGA hardware blocks. Class constructors are used to recursively generate the represented logic structure, including all contained sub-components. The primitive building blocks are wires and gates, which can be used to construct more complex objects. Since, JHDL describes circuit structure, application re-coding is required, if the source is an algorithmic description.

JHDL maintains an open area pool on the device of which parts can be dynamically allocated to implement new objects. The authors introduce explicit destruction of objects (instead of using a garbage collector) to free areas and hand them back to the area pool. JHDL provides reconfigurable elements, called `PRSocket`, which can receive a `Reconfigure(int)` call, requesting a specified implementation. Depending on the argument, new circuit nodes are created:

```
1 class myConfigGroup extends ConfigGroup {
2   ...
3   /* A Node is the base class for all JHDL Logic */
4   Node getNewCircuit( int id, PRSocket sock ){
5     switch(id){ case 1: return new Circuit1( ... );
6                 case 2: return new Circuit2( ... );
7                 case 3: return new Circuit3( ... );
8     ... } } }
```

Reconfiguration control is expressed explicitly, in this case the implemented logic depends on the argument `n`:

```
PRSocket.Reconfigure( n )
```

The approach provides powerful simulation abilities, including hardware-in-the-loop analysis. However, clocks need to be halted during reconfiguration and there is no specification for reconfiguration times. This prohibits simulation with accurate timing.

A synthesis framework for JHDL and Xilinx JBits has also been published[44, 98].

3.2.3 VHDL Based Approaches

Pebble / Quartz

Modeling	Level of abstraction	Register transfer
	Expression of DPR requests	Fixed schedule or runtime condition, implicit
	Dynamic state persistency	None
	Reconfiguration control	Software
	Multiple reconfigurable areas	No/Not shown
	Shared use of reconfigurable areas	No/Not shown
Simulation (yes)	Accuracy	Weak
	Reconfiguration overhead	Omitted
	Persistency overhead	Does not apply
	Scheduling overhead	Does not apply
Synthesis (yes)	Tool support	Limited

Parameterized Block Language (Pebble)[73] is a HDL based on structural VHDL. It is intended to automate the generation of parameterized circuits that can be exchanged at runtime. Pebble provides several ways to describe runtime reconfiguration.

One is based on a mux/demux encapsulation of logic variants [75, 119]. Alternatives are enclosed in `RC_DMux` and `RC_Mux` demultiplexer and multiplexer blocks. Only one of the alternatives is configured to hardware at runtime. The select inputs of `RC_DMux` and `RC_Mux` are used as reconfiguration conditions. Such an application of the *partial evaluation* technique was demonstrated in [74] in more detail.

Another way of describing runtime reconfiguration is by using specialized control flow statements[38]. The `RECONFIGURE` statement is combined with an `IF` block, where the condition is used to trigger a reconfiguration for the statement body. Alternatively, a `RECONFIGURE FOR` is used to dynamically adapt a `FOR` loop implementation to a bound, which is known at runtime only.

Pebble was extended to RTPebble[38, 37], which provides run-time parameterizable description blocks. According to a parameter change the circuit logic is being modified.

The specification does not cover reconfiguration times. There was no persistency concept demonstrated in literature, the reconfigurable blocks seem to be stateless.

A compiled model can be simulated using the Rebecca simulator. The authors demonstrated synthesis for an Xilinx 6200 FPGA [73]. In a second demonstration, the compilation of the Pebble code was performed by the *run-time parameterizable compiler*. The reconfiguration control was done using hand-written code on a processor on a host PC. The PC was equipped with a Celoxica RC-1000 PCI card containing a Xilinx Virtex XCV 1000 chip.

With the experiences of Pebble in mind, the Quartz language[54, 94] was designed. It allows descriptions at a higher level of abstraction including higher order functions or overloading . However, the features to express runtime reconfiguration are the same as in Pebble[93].

DCS

Modeling	Level of abstraction	Register transfer
	Expression of DPR requests	Runtime condition
	Dynamic state persistency	Manual
	Reconfiguration control	Hardware
	Multiple reconfigurable areas	Online scheduled
	Shared use of reconfigurable areas	Online scheduled
Simulation (yes)	Accuracy	Cycle accurate
	Reconfiguration overhead	Included
	Persistency overhead	Included
	Scheduling overhead	Included
Synthesis (yes)	Tool support	Full

The *Digital Circuit Switching* (DCS)[124, 114, 111, 113, 112, 109] is one of the more powerful and comprehensive approaches published so far. It was initially intended to enable simulative verification of reconfigurable circuits, but was then extended to allow for synthesis as well. While VHDL is used to describe a static version of the design, an additional RIF file[107] contains reconfiguration information for the VHDL tasks. This includes boolean activation and deactivation conditions, information about mutual exclusive tasks and geometric relations.

The *DCSTech* tool implements the VHDL design according to the specifications inside the RIF file. Resulting implementation files for the reconfigurable instances can be further processed with standard EDA tools to generate timing information (in terms of SDF files) and a gate level VITAL model.

These files and the modified design may then be processed by *DCSSim*, generating a timing accurate simulation model of the design, including the reconfiguration behavior. Unfortunately, this simulation needs to be performed on the gate level model.

DCS provides mechanisms for dynamic state persistency in two ways[108]. Firstly, saving a task's state on displacement and restoring it later on is possible. However, this seems to involve some manually coded interaction with the preemption interface of the reconfiguration controller. Secondly, passing a task's state to its direct successor in the same reconfigurable area is possible, if the underlying FPGA technology supports it

(e.g. Xilinx XC6200). This requires some additional effort in the backend flow and controlling the placement of sequential elements.

DCSTech is also capable of generating a reconfiguration controller hardware by reading the RIF file[110]. This makes the synthesis abilities of the DCS flow fairly complete. Since, its development started at a time where FPGA vendor tools were less mature, it also performs parts of their tasks, e.g. Xilinx Modular Flow[107]. Along with this comes a dependency on certain device families, e.g. Xilinx XC6200 or (later) Xilinx Virtex.

3.2.4 Other Approaches

Ptolemy II

The Ptolemy II [66] project is a framework for integration of multiple models of computation. Among others, it features a mutable model of computation, which fits to FPGA targets. There are works to integrate JHDL into the Ptolemy II framework [139].

The Ptolemy II framework[86, 66] allows simulating multiple models of computation. The models consist of actors, which may use data and type polymorphism to model dynamic reconfiguration. There was also work to integrate JHDL into the Ptolemy framework[139].

Lava

Modeling	Level of abstraction	Structural register transfer
	Expression of DPR requests	Implicit
	Dynamic state persistency	None
	Reconfiguration control	Hardware
	Multiple reconfigurable areas	Not shown
	Shared use of reconfigurable areas	No
Simulation (yes)	Accuracy	Weak
	Reconfiguration overhead	Omitted
	Persistency overhead	Does not apply
	Scheduling overhead	Does not apply
Synthesis (no)	Tool support	None

Lava[21] is a language based on the pure functional programming language Haskell. It aims at providing a hardware description that can be simulated and allows proofs of circuit properties. The language provides a bit type and an integer type, as well as operators on them. These operators are circuit building blocks, for example `and` gates or arithmetic operators. Circuit primitives are assembled using serial and parallel composition. Like JHDL, these components can be aggregated to larger ones in a bottom-up manner. A strong type system ensures compatibility in this composition. Another similarity to JHDL is the expression of time: It is measured in steps of clock-high and clock-low values and does not have a notion of duration.

Lava was extended [121, 122] to support modeling of partial reconfiguration. The clock of non-modified regions is assumed to be halted during reconfiguration, so it does not fully describe dynamic reconfiguration.

Circuit specialization is implemented as a choice of types. In Lava, circuit elements are described as functions. Haskell functions have a type signature, which is checked by the compiler. The type signature of a reconfigurable circuit offers alternatives, that is to say, it may accept different arguments and yield results of different types. The selection of input type determines the output type. Unused alternatives may be discarded, which expresses circuit specialization.

Reconfiguration is modelled on top of this. An additional input is used to select among the type alternatives. By providing different values to this select input specialization turns into dynamic replacement.

The use of multiple reconfigurable areas is not demonstrated but should be possible with the presented approach. Shared use of an area is not supported, it would be unfolded into the application logic. The authors in both works demonstrated simulation abilities but no synthesis or integration into industrial tool flows.

3.3 Classification of OSSS+R

Modeling	Level of abstraction	Application
	Expression of DPR requests	Implicit
	Dynamic state persistency	Manual or automatic
	Reconfiguration control	Hardware
	Multiple reconfigurable areas	Online scheduled
	Shared use of reconfigurable areas	Online scheduled
Simulation (yes)	Accuracy	Cycle accurate
	Reconfiguration overhead	Included
	Persistency overhead	Included
	Scheduling overhead	Included
Synthesis (yes)	Tool support	Full

OSSS+R is a SystemC based approach, where the reconfiguration infrastructure is not interwoven with the SystemC module hierarchy. This allows changes on the aspects of reconfiguration without touching the module hierarchy (which is time-consuming to do). The input expresses reconfiguration in terms of object manipulation, which in turn implicitly determines reconfiguration aspects. An object's state is automatically preserved in the case of permanent contexts and needs to be done manually if the designer decides to use the more low-level temporary contexts. This takes the burden of preparatory tasks off the designer.

Reconfiguration control is done in hardware. The application dependent part of this control unit is automatically generated. It supports multiple reconfigurable areas and schedules reconfigurations at runtime. Making decisions at runtime allows the most flexibility. Still, the designer could specify a static schedule for its application and let the schedulers follow a fixed sequence.

Access control to reconfigurable areas is also handled implicitly and scheduled at runtime. In case of named contexts, a very flexible sharing of reconfigurable ressources between different processes is possible. Sharing reconfigurable resources may increase area utilization for some applications.

The system can be simulated in a cycle accurate manner, including all communication, state preservation and reconfiguration overhead. Latency and throughput issues may be investigated in the abstract model, once precise reconfiguration times are back-annotated. A synthesis tool is capable of translating the model to VHDL. If needed, this model can be generated to emulate partial reconfiguration using multiplexers. The resulting VHDL simulation model can be processed by standard simulators to validate synthesis results.

3.4 Overview of the Approaches

		Dynamic module library	ReChannel	UF Campina Grande	SystemC DRCF	SyCERS	SystemC' mantic	PERFECTO	ITI Lübeck	OCAPI-xl
Modeling	Expression of DPR requests	C	E	FC	C	C	FC	E	FC	I
	Dynamic state persistency	-	M	Z	Z	-	-	-	M	Z
	Reconfiguration control	-	HS	H	-	H	-	-	S	HS
	Multiple reconfigurable areas	-	-	-	-	-	-	-	OS	OS
	Shared use of reconfigurable areas	-	-	-	-	-	-	OS	OS	OS
Simulation	Accuracy	W	CA	CA	W	CT	CT	CA	CA	W
	Reconfiguration overhead	O	I	I	O	I	I	I	I	O
	Persistency overhead	-	I	O	O	-	-	-	-	O
	Scheduling overhead	-	-	-	-	-	-	I	I	I
	Tool support (Synthesis)	-	-	-	-	-	-	-	L	L

		VPRS	Tiles	RT-C	JHDL	Pebble/Quartz	DCS	Lava	OSSS+R
Modeling	Expression of DPR requests	E	-	I	E	FC/I	C	I	I
	Dynamic state persistency	-	-	-	M	-	M	-	MA
	Reconfiguration control	-	-	S	S	S	H	H	H
	Multiple reconfigurable areas	-	-	OS	OS	-	OS	-	OS
	Shared use of reconfigurable areas	-	-	OS	-	-	OS	-	OS
Simulation	Accuracy	CA	-	-	W	W	CA	W	CA
	Reconfiguration overhead	I	-	-	O	O	I	O	I
	Persistency overhead	-	-	-	I	-	I	-	I
	Scheduling overhead	-	-	-	-	-	I	-	I
	Tool support (Synthesis)	-	-	L	LF	L	F	-	F

Table 3.1: Properties of related approaches

Table 3.1 gives an overview of the related work. The legend is given in table 3.2.

Symbol	Meaning	Symbol	Meaning
C	Runtime condition	E	Explicit
FC	Fixed schedule or runtime condition	I	Implicit
Z	Zero-time	H	Hardware
O	Omitted	S	Software
I	Included	HS	Hardware or software
M	Manual	CA	Cycle accurate
MA	Manual or automatic	CT	Coarse timed
		W	Weak
L	Limited	OS	Online scheduled
LF	Limited or full	-	No / None / Not shown / Does not apply

Table 3.2: Properties of related approaches, symbol legend

3.5 Chapter Summary

In this chapter, classification scheme for related works was introduced and applied to several approaches. The scheme covers modeling, simulation and synthesis aspects. The selected approaches focus on representing reconfiguration aspects in their entry model. Finally, the proposed approach (OSSS+R) was classified using the same scheme.

Chapter 4

OSSS+R Modeling

In this chapter, the proposed modeling approach is presented. First, the design flow is discussed. Then an explanation of the theoretical background for modeling dynamic partial reconfiguration is given. It is followed by a description of the proposed modeling using objects and dynamic scheduling. Finally, a set of optimization aids is introduced to tune an application's behavior to the designer's needs.

4.1 Design Flow

The proposed design flow is called *OSSS+R*. As the name suggests, it is based on OSSS (see Section 2.5). In Chapter 1, four goals of this work were introduced: integration into existing tool flows, a short development cycle, design quality, and platform independence. The first of these goals is addressed by using C/C++ and SystemC as design entry and VHDL, as a link to industrial back-end tools. To be precise, the design entry may be either a plain C++ specification, which is to be transformed into a SystemC design or a SystemC design, which is to be extended by algorithms implemented in C++. Since, non-synthesizable SystemC covers all of C++, the main task is to move the C++ functionality into a hierarchy of SystemC modules and processes. Here, dynamic memory management, pointers, and all possible datatypes are permitted. Thread spawning, file I/O or even graphical user interfaces are possible.

Since, this plethora of elements in such a model cannot be automatically transformed into a synthesizable representation, the designer needs to reduce the features to a reasonable set of features with known hardware semantics. This task is easier, if the designer has suitable replacements for non-synthesizable design elements at hand. Some of them are described in Section 2.5.

Figure 4.1 gives an overview on the design flow and the described transformation step in particular. After transformation, the model is represented in OSSS. It may already contain reconfigurable elements provided by OSSS+R, but this is not necessary. These can be introduced later on and are described in Section 4.3.

The designer may perform design space exploration on the OSSS+R model. The exploration step is not done automatically by any tool but simplified by OSSS+R. Reconfigurable areas may be added and functionalities may be mapped to them. Polymorphic pointers or objects being used mostly mutually exclusive make good starting points. It is easy to add and remove reconfiguration abilities for these candidate objects.

Note, that the philosophy is *not* to tailor an application to partial reconfiguration when moving from C++ code to synthesizable SystemC. This would impose much work if partial reconfiguration turns out to be infeasible or to offer too little advantage in the end. Instead, the suggestion is to pick low-hanging fruits and spend little effort in this phase. Some C++ models contain structures that can be easily transformed into a

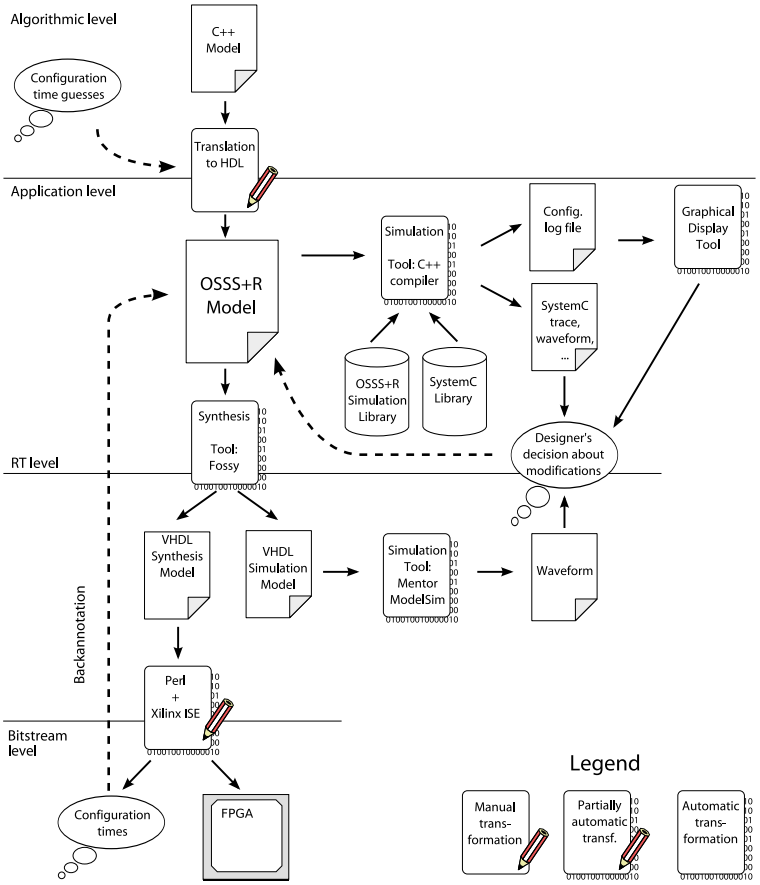

Figure 4.1: Detailed tool flow for OSSS+R

reconfigurable implementation using OSSS+R. This will be explained in further detail inside this chapter.

Once an OSSS+R model is found to be worth exploring, it can be simulated. This is done using a SystemC simulation library, for example the OSCI reference implementation. OSSS+R elements are added via an extra library. In SystemC terminology, OSSS+R acts as an domain specific extension library. During this simulation, trace files using SystemC *sc_trace* concept may be used. Additionally, a log file can be generated that records all events relevant for partial configuration. Method executions, scheduling phases, reconfigurations, and many other kinds of events are stored in that file. It can be inspected using an external tool [51] that visualizes events over a time axis. This allows to seek unbalanced utilization, trashing effects and other unwanted situations. Having the graphic visualization and the SystemC traces at hand, the designer can implement design modifications to optimize the model. This can result in a different number of reconfigurable areas or a modified mapping of functionality to these or even more sophisticated optimizations. OSSS+R offers several modeling elements targeted against specific problems that may occur. These are introduced in Section 4.4.

If the application model is found acceptable, it is to be implemented. Although SystemC is not specifically designed for automated synthesis [29], this does not mean that the model is to be re-coded manually on a lower level of abstraction. Instead an OSSS synthesis tool, *fossy* [47] automatically transforms the model to register-transfer level VHDL code. This includes an implicit-to-explicit finite-state machine transformation.

There are two models that can be generated: A synthesis model and a simulation model. Since, partial reconfiguration cannot be expressed directly in VHDL (or other common HDLs like Verilog), simulation would be difficult. To simulate the synthesis model, manipulations of the FPGAs configuration port during reconfiguration would need to show an effect. Uploading a bitstream to these pins would require the simulator to modify the model, for example to exchange architectures of a component during the simulation run. Or, if the model is synthesized down to a set of configuration bitstreams, a comprehensive platform model could implement the FPGA device's behavior during reconfiguration. Neither of these two approaches are feasible.

Instead, the simulation model can be seen as a superset of the synthesis model. It contains all components of the synthesis model plus some extra components that mimic reconfiguration. This model does not rely on platform specific features and is accepted by industrial VHDL simulators, for example Mentor Graphics ModelSim[9].

Both simulation and synthesis model may be generated as plain SystemC models, too. This is useful, if a VHDL/SystemC co-simulation is to be avoided.

The synthesis model contains all platform independent parts of the model. This includes the application logic, schedulers, variants for reconfigurable regions, controllers etc. Since, it is intended to run on a real physical platform, there are platform specific components required. These cannot be generated by a platform-independent tool by definition. Instead, a platform support package needs to provide register transfer level components to allow low-level control of the platform's reconfiguration resources. In order to keep the methodology platform-independent, a very limited set of platform features is required. Additionally, the interface between the platform dependent and platform independent part is designed to be slim and generic. It consists of a handshaking scheme to request only specific bitstreams.

Back-end synthesis and bitstream generation tools require specific properties for their inputs. In case of Xilinx tools, which are used to demonstrate the feasibility of this approach, this includes a process-free top level design component and inference of bus macros. This can be done manually with partial automation using scripts. For the experiments, this was done using Perl[6]. Meanwhile, there are more sophisticated research tools for such purposes, like Part-E[39, 40].

The back-end phase is completed when bitstream files are generated. These can be used to configure the platform. Additionally, they can be used to calculate the

exact duration of each configuration process, depending on the bitstream size. This can be back-annotated to the OSSS+R model. Initially, the OSSS+R model required the designer to make rough estimates based on compilations of non-optimized designs or experience. Now, these can be replaced by accurate values.

Both simulation phases (the OSSS+R model and the register-transfer VHDL simulation model) are cycle-accurate. This offers an important opportunity for the designer: If the final implementation shows unexpected behavior (e.g. buffer underruns), a timing analysis is possible using the abstract OSSS+R model. It is not necessary to track down performance bottlenecks or latency problems due to runtime reconfiguration at a low level of abstraction.

4.2 Objects and Modules

When modeling reconfigurable hardware, two kinds of limited resources exist: reconfigurable areas on a device and resources required to manipulate the configurations. When it comes to reconfigurable areas, the primary requirement is rather to describe the different variants of such an area than representing the area as such. Using SystemC, two different strategies are obvious: Treating these as C++[125] objects or as SystemC modules.

All approaches presented in Section 3.2.1 treat reconfigurable areas as SystemC modules. In some concepts, it is even required that these modules act as members of a common on-chip bus. On one hand, modules allow a large degree of expressiveness. It is possible to embed active components in terms of threads and clock-driven methods in the modules. And, using modules is natural for a hardware designer. On the other hand, modules do not just allow a great expressiveness, they do require expressing many details:

- There is no common way to specify how to halt and continue a module's internal processing.
- There is no common way to save a module's state including those of all active components. The same applies to the state restauration later on.
- Switching communication is usually done via multiplexers or bus addressing. These multiplexers and busses have to be specified and inferred.

In previously published approaches, these issues are tackled only partially. The designer often needs to manually specify start and stop of processing. State preservation is omitted or to be implemented manually. And even the use of multiplexers or certain busses to model switching imposes some further consideration by the designer.

Treating reconfigurable areas as C++ objects has some disadvantages, too. Firstly, objects are passive since no processes can be embedded. This prohibits the use of SystemC threads or methods. A second drawback is the lack of a demand interface. If a member method execution requires additional data, C++ programmers use pointers to query external objects. Without having pointer support inside the hardware description language, a different mechanism is required.

On the other hand, objects offer some severe advantages. The absence of active components inside is beneficial: In between two method executions the state of an object is solely determined by the member attributes. Expressing saving and restauration is well-defined by using assignment operators. The only remaining issue is switching communication from one to another. Here, C++ provides two usable mechanisms: Mutual exclusive use and polymorphism.

Mutual exclusive use is the more abstract way to express a switch. Using one object implicitly expresses disabling every other object which cannot be provided in conjunction to the new one. This kind of abstraction is used to a large extent in software for modern platforms by means of virtual memory. Expressing DPR this way, the mutual exclusive objects can be considered as virtual hardware[96] objects.

Another way of expressing a switch in hardware is by using polymorphism. In C++, polymorphism means the ability to handle object instances of different types using a single interface. An identifier is associated with a static pointer type, that means, the type is fixed at a compile time. At runtime, however, it may point at varying objects with different types. As a restriction, the types of these objects being pointed at need to be identical or derived from the type of the pointer. Section 2.2.2 clarifies this in more detail. For runtime reconfiguration, polymorphism may be used to refer to a single object, which changes its runtime type. This mimics changing the implementation for an object during runtime while a defined interface is preserved.

In OSSS+R, the object representation for reconfigurable areas was chosen. As explained in Section 2.1, the greater expressiveness of modules is considered less important in comparison to the ease of use.

4.3 Polymorphism and Runtime Reconfiguration

4.3.1 Reconfigurable Objects

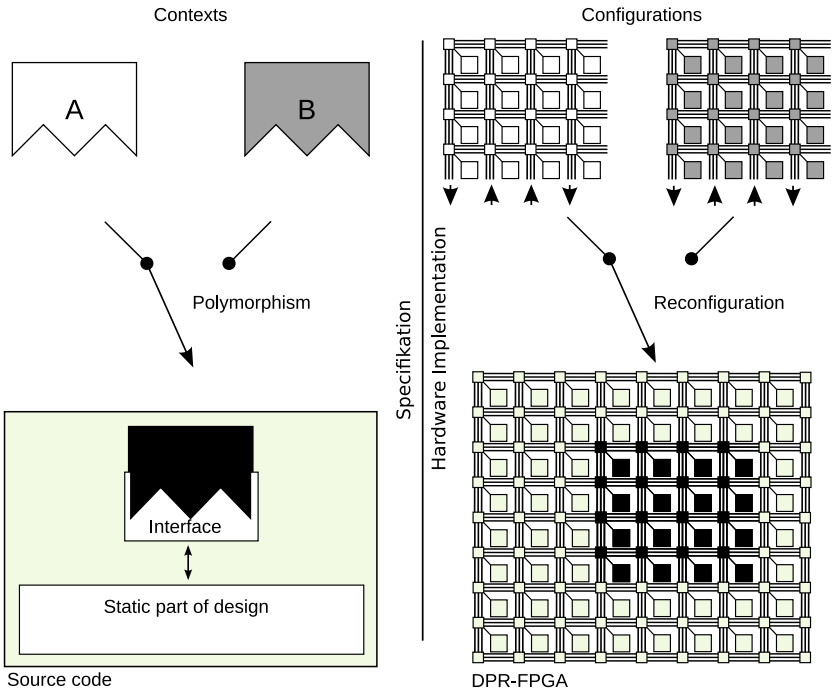

Figure 4.2: Analogy: Polymorphism and dynamic partial reconfiguration

Having polymorphism available in a hardware description language makes it worth comparing the properties of this concept to the demands of reconfigurable hardware implementations. Figure 4.2 shows the analogy between polymorphism and DPR. On the left hand side, the specification contains a static design part relying on a fixed interface to the different objects. This shows polymorphism, since the interface can be connected to instances of type A and B. On the right hand side, the implementation requirements are shown. A reconfigurable area of the FPGA is shown in black. Since, the static environment of this area does not change by definition, it requires an fixed interface on signal level. Values originating from the reconfigurable area may be processed at different locations in the static area and in different ways. A signal value's

interpretation may vary, while the electrical interface (indicated by the small arrows below the configuration icons) stays the same.

However, this cannot be expressed using a polymorphic *value* type. Reconfigurable areas are limited resources with an unlimited lifetime, making them entity objects.

Definition 12 - ***Reconfigurable object****:*
A reconfigurable object is a shared entity, which represents a reconfigurable area.

Definition 13 - ***Anonymous access object****:*
The anonymous access object is an instance, which is used to manipulate the current FPGA context of a reconfigurable object.

The anonymous access object allows invoking methods (including copy operations) on a reconfigurable area. It is a programming language construct to be used in expressions like method invocations and assignment. It is assignment compatible with polymorphic objects of the same interface type.

The reconfigurable object solely represents the hardware entity. The formal separation of the reconfigurable object and anonymous access object eases clearly defining extensions to the model of reconfigurable objects in the following sections of this thesis. The reason for the term *anonymous* will be clarified as well.

Access control. A reconfigurable object is an entity object, since it abstracts from a physical resource. Since this resource is limited, it is desirable to maximize its utilization at runtime. Therefore, anonymous access objects can be manipulated by multiple user processes. This concept of sharedness between processes is borrowed from the shared objects as introduced by the ODETTE project. The set of potential user processes is defined as $\mathcal{U} := u_0, .., u_n$. Each u is a user-defined clocked process with an implicit state machine (SystemC `SC_CTHREAD`). The set of all reconfigurable objects is defined as $\mathcal{R} := r_0, .., r_n$. For each r, a set of user processes is defined as $\mathcal{U}_r \subset \mathcal{U}$. Two cases are distinguished, depending on the number of user processes: If only one user-defined process contains access statements to the reconfigurable object r ($|\mathcal{U}_r| = 1$), the process holds a permanent permission to manipulate the anonymous access object. If there are at least two user processes containing manipulation statements for the same temporary context, an access arbitration mechanism is used. The access arbitration requires additional communication between the user-defined processes and the access control, which in turn may alter the timing.

Besides avoiding access conflicts, the access control unit has further purposes. First, it handles reconfiguration requests from the user-defined processes. Second, it keeps track of the logic configured to the dynamic areas. This allows to hide superfluous reconfigurations, overwriting logic with an identical one.

Access scheduler. The arbitration mechanism is called access *scheduler*, since it may maintain a memory to memorize past events or even do planning for the future. It is called, whenever the reconfigurable object is ready to serve a request and at least one request by a user process is detected. This explicitly includes calling the scheduler for a single request. The reason is that some history-based arbitration algorithms like Round Robin, do require to memorise the last granted access. The access scheduler is discussed in Section 4.3.3 in greater detail.

One design decision was to have a runtime scheduling instead of always following a fixed schedule. The primary reason is that it better fits into the nature of the system description. Inside the processes, it is well possible to influence the control flow (branch

conditions and loop iterations) by reading module's ports. This is likely to influence the call profile for reconfigurable objects regarding which methods are invoked when. For the reconfigurable resource, this is a dynamic demand profile which can be best served using a runtime-calculated schedule. Besides the demand for flexibility and its price in terms of scheduling overhead, a runtime-schedule may lead to faster execution. A runtime schedule may take the history of requests into account to improve the quality of decisions. This is especially important in the field of partial reconfiguration, since reconfiguration times may significantly contribute to the application runtime. In other words, poor scheduling decisions resulting in more frequent object and class switches are likely to be very costly.

However, for hard real-time constrained systems, a fixed schedule is mandatory. A fixed schedule is still possible by pre-calculating a decision table and using a dynamic scheduler, which performs look-ups in this table. Such systems are possible with OSSS+R but are not in its focus. Instead, systems which are to be optimized for a fast average-case execution are targeted.

Manipulation, interface and implementation classes. Along with a reconfigurable object, an interface class t_r needs to be declared. This relates to the interface class of a polymorphic value object or the static type of a C++ polymorphic pointer. The class serves two purposes:

- It restricts the set of possible implementation classes to those being identical or derived from the interface class. This $\lhd^*$ relation is required to statically check type conformance of assignments to the anonymous access object.
- It defines the interface usable for object manipulation. All public methods defined in t_r can be invoked on the anonymous access object.

Instantiation. Since reconfigurable objects are entity objects, they can only be instantiated as members of modules.

Sharedness. Reconfigurable objects are *shared* in the sense of being usable by multiple clocked processes. Although this can be used to express interprocess communication (like shared objects do), this is not the primary purpose of shared accesses. Like shared objects (as introduced before), reconfigurable objects represent a shared datapath. Reconfigurable areas can be considered a very limited and valuable resource. Therefore, allowing shared access to this resource may enhance their utilization.

Invalid states. Reconfigurable objects cannot be brought into an invalid state during normal system operation. The reason is that there is no way to express manipulations of the anonymous access object to achieve this. C++ pointers can be obsolete (or *dangling*), if the target memory location no longer represents a valid object of the correct class. Additionally, C++ pointers can be manipulated directly by circumventing all type checks, rendering them completely unpredictable. Finally, one may assign the special value `NULL` to them, making them explicitly point at no object at all. All these pitfalls cannot be expressed with anonymous access objects. Besides hardware failure, only a system reset or initial power-up can invalidate a reconfigurable object.

Class and Object Switch

In the proposed approach, a separation between the configuration of a FPGA area and its dynamic state is an essential element. With respect to C++ modeling, this is the separation of class type and the data attribute values of a pointer's target. It can be expressed as follows [1]:

[1] Memory leaks and missing object destructions in the code snippet are ignored for simplicity.

```
1  A * poly = new A();
2
3  A a;
4  B b;
5
6  poly = &a;                  // object switch
7  poly = &b;                  // object and class switch
8  poly = new (poly) A();      // class switch ("placement new")
9  poly = new (poly) A();      // neither object or class switch
10 poly->method();             // neither object or class switch
```

The assignment in line 6 makes `poly` point at `a`, which is a different object than the previous destination of `poly`. Therefore, an *object switch* occurred. The next assignment additionally contains a *class switch* because the type of object which `poly` points at is changing, too. It switches from class `A` to `B`. The third assignment in line 8 generates a new object at the same memory location as before, but with a different class. Finally, line 9 and 10 preserve both class type and address, although the data in memory is modified. For C++, we define two kinds of switches:

Definition 14 - ***Class switch (C++)**:*
A class switch changes the runtime type of a polymorphic pointer's target, this may include object switch.

Definition 15 - ***Object switch (C++)**:*
An object switch makes polymorphic pointer refer to an object with a different identifier.

Please take note that there may be objects without identifiers in C++. One example is given in the right hand side of line 1 in the source code example. For the definitions given above, two objects without identifiers are treated like having different identifiers, iff they are implemented at different memory addresses.

As described before, the hardware implementation of polymorphism does not separate between a pointer and its destination. It rather knows objects of polymorphic nature *containing* values of different types. Therefore, slightly different definitions of object and class switch for hardware implementation are needed:

Definition 16 - ***Class switch (Hardware)**:*
A class switch is a change in the dynamic type of a temporary context.

Definition 17 - ***Object switch (Hardware)**:*
An object switch makes a polymorphic object contain an object of a different identifier.

Note, that the term *polymorphic* is used as an attribute in this definition. Thus the definition can be applied to any object of polymorphic nature, such as polymorphic value objects or reconfigurable objects.

This separation of object and class switch is inspired by a definition given by Noguera and Badia[87, 88]. The definitions do not match exactly, in particular for the object switch. In Noguera and Badia's definition, object switches are determined by a change in the object's attributes. Since, executing methods that alter the object's state, these are treated as object switches in Noguera and Badia's definition.

4.3.2 Contexts

Permanent Contexts

When applying the terms object switch and class switch to the possible manipulation by using the anonymous access object, it turns out that no object switch can be expressed. There is only one object mapped to the physical resource, since the very same *identifier* is used in all cases. Having object switches would require a situation, where different objects (and different identifiers) would be used.

The anonymous access object always reflects the current state of the very same reconfigurable area. The relation is comparable to a C++ pointer, which always refers to the very same memory location.

While it was already shown that this is sufficient to express polymorphism, it makes expressing mutual exclusive use of different objects very cumbersome.

A first observation is that objects in a C/C++ application are value objects. Mapping several of these to a reconfigurable object requires additional effort. If one object is to be replaced by another one, the state of the current one is to be saved. This needs to be done by assigning the anonymous access object to a polymorphic object of the same interface type (which only acts as a storage object). Then, the new state is to be set by assigning it to the reconfigurable object. Finally, the access can be made. This is a management task, which is error-prone and cumbersome to implement manually. In C++, this could be done as follows:

```
1  A * poly = new A(); // polymorphic pointer
2
3  A obj_1;
4  A obj_2;
5
6  // first access
7  *poly = obj_1; // copy values of obj_1 to poly's destin.
8  poly->foo();   // operate
9
10 // second access
11 obj_1 = *poly; // save back values into obj_1
12 *poly = obj_2;
13 poly->bar();
```

This code even assumes that no class switch is to be described, which would make the code even more complicated.

It is likely that the application level code just reads:

```
1  A obj_1;
2  A obj_2;
3
4  obj_1->foo();
5  obj_2->bar();
```

It is easy to see that requiring the designer to manually translate its accesses to exploit polymorphism is not a suitable solution. In contrast, the solution should allow just to *name* the desired object on which an operation is to be performed. This already clearly determines, which steps to perform, in order to map it to a polymorphic resource.

A second problem arises from the interface class. Since, the interface class is static, it cannot adopt to the current application level object to implement. This can be tolerated, as long as the application objects to be used mutually exclusive share a suitable common base class, which can be used to interface the reconfigurable object. If this is not the case, then the *union* of all required interfaces needs to be created by means of a new class, which needs to be inserted in the class inheritance tree. This would

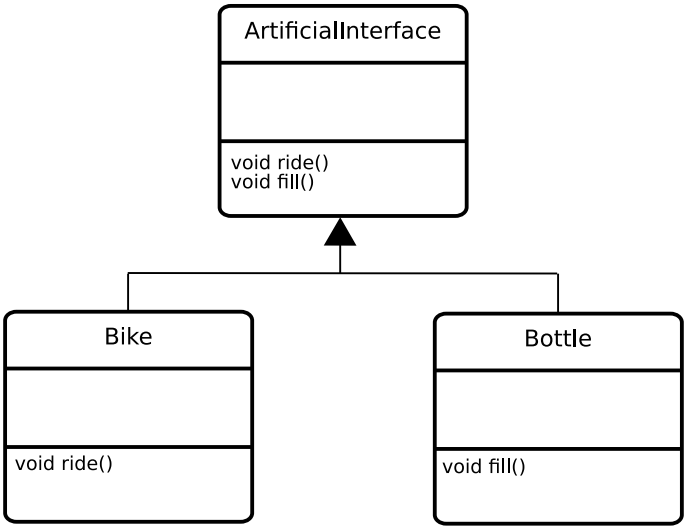

Figure 4.3: Artificial interface class

violate the principles of object-oriented design. Class relations would be drawn, which are completely artificial. Figure 4.3 illustrates such a situation.

The proposed solution is to explicitly instantiate the objects to be used mutually exclusive and refer to them by name. These objects describe FPGA contexts as defined in Section 2.6. They must not interfere with each other from the application's point of view. In other words, manipulating one of these objects may not affect others. Even though they are to be implemented on a reconfigurable object (and therefore share implementation resources) they must be permanent.

Definition 18 - ***Permanent context****:*
A permanent context is a FPGA context which is accessible using an identifier.

Definition 19 - ***Context enabling****:*
Enabling a context is the process of making it accessible through the reconfigurable object.

Other approaches use the term *activation* instead of *enabling* for this process. In other publications of our working group, *active* and *passive* are used to determine whether an object has its own thread of control. This thesis follows the same terminology, for example, SystemC modules are active, since they may contain threads. Reconfigurable objects and their contexts are passive.

It is possible to automatically deduce the sequence of steps required to perform manipulations of permanent contexts on reconfigurable objects, hiding them from the designer. This is provided by permanent contexts.

The anonymous access object always reflects to the current state of the reconfigurable area. Permanent contexts refer to application level objects mapped to the reconfigurable object in a time-sharing manner. Manipulating a permanent context requires enabling it, which is a modification of the reconfigurable area. So the anonymous access object and the permanent context may interact. As a consequence, there are two ways to access a context.

Definition 20 - ***Anonymous access**:*
Anonymous access is a manipulation performed on the anonymous access object. It is possible to manipulate permanent contexts this way without explicitly using their identifiers.

Definition 21 - ***Named access**:*
Named access is a manipulation performed on a permanent context. The permanent context is indicated by its identifier.

The separation of anonymous access and named access allows to use two different levels of abstraction, depending on the designer's needs. If a low-level control is advisable, the anonymous access should be used. On the other hand, the named access allows more abstraction and is more comfortable to use in more complex scenarios.

Like the anonymous access object, permanent contexts are polymorphic. They are equipped with an interface class, serving the same two purposes as the anonymous access object interface class: runtime type restriction and interface definition. There is a type restriction for this interface class as well. Let t_r be the interface class of the reconfigurable object (that means, used by the anonymous access object), t_c the interface class of a permanent context. It is required that $t_r \lhd^* t_c$. This ensures that the interface provided by t_r is supported by all t_c. In other words, the interface type used for anonymous access is compatible to all permanent contexts.

The requirement $t_r \lhd^* t_c$ ensures type correctness for all method invocations except assignments. To illustrate this let $t_{assign,r}$ be the type of an object, which is being assigned to the anonymous access object (assignment source) and $t_{assign,c}$ the type of an object which is being assigned to a permanent context. Assignments *to* the anonymous access object are abbreviated *anonymous assignments* for brevity.

The runtime type restrictions require $t_r \lhd^* t_c \lhd^* t_{assign,c}$ for assignments to permanent contexts. For anonymous assignments, we have $t_r \lhd^* t_{assign,r}$ only. If a permanent context is currently enabled, the condition $t_c \lhd^* t_{assign,r}$ needs to hold, too. Sadly, anonymous assignments do not guarantee this condition!

One way to handle such dangerous anonymous assignment statements would be to forbid them or at least issue a warning during simulation or synthesis. While this is a safe way to handle the situation, it is pretty restrictive. The proposed approach treats anonymous assignments as a request to create a temporary context. If a permanent context is currently enabled, it is being disabled. This includes saving its state, therefore the anonymous assignment does not touch it. Now, a non-permanent context, the *temporary context* is created by executing the assignment on the reconfigurable area. Whenever this temporary context is to be disabled due to enabling a permanent context, its state is lost. The rationale is that manipulations performed on the anonymous access object are treated as manipulations of the reconfigurable area, which in turn changes whenever permanent contexts are used. In the case the designer needs to make sure no temporary context is created at any time, it can be easily checked by a synthesis tool.

Definition 22 - ***Temporary context**:*
The temporary context is a non-permanent FPGA context. It is created by anonymous assignments and destructed by use of permanent contexts.

Permanent contexts allow a modeling style, which can be seen as use of *virtual hardware* [96]. A similar concept for data storage is well-known to software programmers as *virtual memory*. Variable declarations in a typical software implementation

do not include specifications of physical resources to be used, that means, the allocation of memory addresses. The operating system serves as an underlying layer, which automatically swaps logical memory from and into physical memory. Permanent contexts permit the same modeling style. They are transparently swapped in and out of the reconfigurable area without requiring the application to take notice.

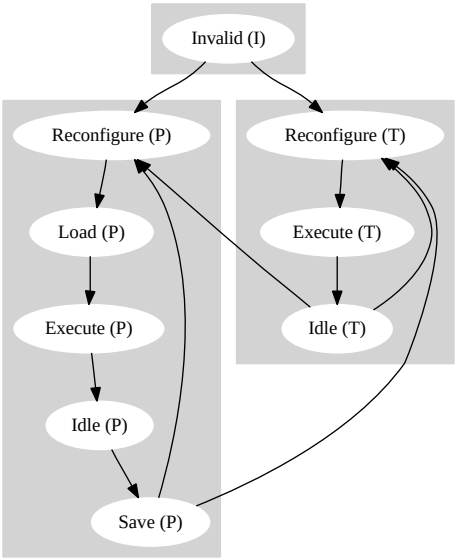

Figure 4.4: State diagram for reconfigurable objects

Figure 4.4 illustrates the relation between permanent contexts and the temporary context. The states are partitioned into three clusters: initial state (I), permanent context (P) and temporary context (T). The states of the P and T clusters exist, if there are permanent contexts or the one temporary context. Otherwise, the states and all ingoing and outgoing edges are omitted.

The system has three steady states, "invalid" and two "idle" states. Object manipulations force transitions until a steady state is reached again. The kind of manipulation determines, which edges are taken and which operations are performed. The parameters are:

- Is an object switch requested?
- Is a class switch requested?
- Is the manipulation an assignment?

The initial state is titled "invalid". Operations other than assignments *to* contexts are illegal in this state. If a permanent context is used in the assignment, the edge to "reconfigure (P)" is taken, otherwise the edge to "reconfigure (T)". The "reconfigure" states handle class switches. In both "reconfigure" states the FPGA configuration is adapted as demanded. In case of a request for a class switch (even if both old and new classes match) a reset of the reconfigurable area is performed. The reset code is determined by the implementation classes' constructor code. In case the current configuration matches (in other words: no class switch is to be performed), no operation is executed.

In the "load" state a permanent context's data members are loaded, if an object switch is requested. As an exception, assignments to contexts cause no operation. It would not make much sense to load data, which is going to be discarded immediately after loading it. When entering any "execute" states, the correct context is guaranteed to be enabled. The system executes any calculations associated by the operation. This is always a method execution. Even assignments from contexts and to contexts are

method executions.

Next, the system reaches one of the "idle" states. It remains in this state, until the next operation is to be performed. The "save" state stores the data member's values of permanent contexts in case of an object switch. Depending on the kind of context the operation is performed on, the "reconfigure" states of the P or T cluster are reached.

Both P and T cluster are identical, except from those states missing in the T cluster which perform object persistency. The "load" and "save" states are omitted in the T cluster.

4.3.3 Access Scheduler

As mentioned before, a scheduler is instantiated, if more than user process is communicating with a reconfigurable object. The implementation of this scheduler is given by a user-defined class type. This class needs to be a refinement of a pre-defined access scheduler base class. This base class provides access to information about the requests. Table 4.1 shows the information provided for each user-defined process.

Additionally to this, a status flag indicates whether another reconfigurable object is

Entry	Explanation
Valid-flag	Indicates whether the user process made a servicable request or not.
Priority	A priority value for the user process.
Save-flag	A flag indicating whether saving the context's dynamic state is required or not.
Save cycles	The number of cycles required to save the state.
Load-flag	A flag indicating whether loading the context's dynamic state is required or not.
Load cycles	The number of cycles required to load the state.
Current class	The internal class identification number for the current class. This identifies the FPGA configuration currently present in the reconfigurable area.
Requested class	The internal class identification number for the requested class. In case of a class switch, this differs from current class.
Configuration cycles	The number of cycles required for a reconfiguration. This field is zero, if no reconfiguration is necessary.
Context identification	Indicates, which context is requested.
Context class	The internal class identification number currently valid for the requested context.

Table 4.1: Request information table available to the access scheduler

currently under reconfiguration. Since the scheduler class may be user-defined, storing additional history information in extra data members of the scheduler class is possible. This permits sophisticated history-aware schedulers.

The scheduler's information table describing the requests contains pre-processed data. It does not indicate which reconfiguration time is required when switching from one class to another for the general case. In contrast, it carries information applied to the current situation of the reconfigurable area (the enabled context identifcation and its class tag). For each request, the scheduler has access to the costs of servicing the request *at the current instant*. Servicing one request may modify the costs for servicing others afterwards. The proposed solution does not require the user-defined scheduler to re-calculate these tables by itself. Doing this calculation requires detailed knowledge of the mechanisms of the permission handling and reconfiguration handling

infrastructure. The scheduler would need to calculate, which steps in Figure 4.4 were to be executed and what the costs would be. To ease the designer's task and circumvent this possible pitfall, the tables are re-calculated automatically.

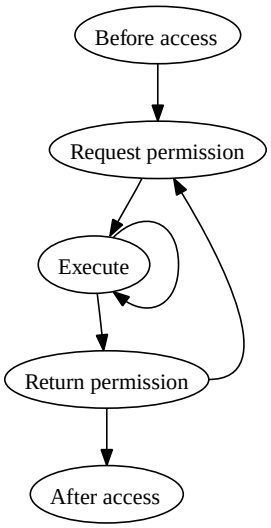

Figure 4.5: State diagram for user processes

Figure 4.5 depicts the possible states of a user process regarding its interaction with the reconfigurable object. Initially, the user process has no permission to manipulate any context. In order to do so, it must request access permission. It may specify which context is to be accessed or use the anonymous access object.

Along with this, the user process may ask for a class switch (see states labeled "reconfigure" in Figure 4.4, too) *for the specified context*. In case of a class switch it may well happen that no reconfiguration is necessary. As a general rule, the user-defined processes may demand class switches for contexts. The reconfigurable object then determines whether the hardware needs to be reconfigured or not.

Once the user process has been granted permission, it may manipulate the reconfigurable area. It may even request further class switches. The permission is always given for one specific context. There is no way a single user process may hold permission for more than one context. Due to this, object switches require returning old permissions. The user process may request the next permission immediately afterwards. Note that requests cannot be canceled and do not time out.

User processes may request class switches and context access permission in separate steps or at once. Access permission requests are the only way a user-defined process can request object switches. As long as requests are pending, the reconfigurable object repeatedly performs the following actions:

1. Resolve anonymous accesses to the currently enabled context. Anonymous assignments are always resolved to the temporary context.
2. Hide non-serviceable requests by marking these as invalid. Non-serviceable requests require access permission, while the previous permission granted by the reconfigurable object was not returned yet.
3. If no serviceable requests are available, wait for some process to return a permission and restart the sequence at the first step.
4. Call the scheduler.
5. If currently enabled context is a permanent one and no class switch was requested, save the context's state.
6. If a class switch is necessary, reconfigure.

7. If a class switch was requested (necessary or not), reset reconfigurable area.
8. If new context is a permanent one and no class switch was requested, load the context's state.
9. If access permission was requested, grant permission.
10. Update internal memories: given grant, configuration, enabled context.

Reconfigurations are skipped, if the reconfigurable area already holds the requested configuration.

The area reset, however is performed even if no reconfiguration was really done. It just depends on the user process requesting a class switch for the context. If a class switch was requested, while a grant was given to the user process, no context state saving and loading is performed. This way the user process may change a context's class and obtain a default-initialized context. Table 4.2 summarizes the handling of dynamic states. Access permissions can be returned at any time. There is only one

	Access permission request only	Access permission and class switch request	Class switch request only
Context save	Yes	No	No
Reset	No	Yes	Yes
Context load	Yes	No	No

Table 4.2: Dynamic state changes upon various requests

access permission for each reconfigurable object[2]. No process can hold multiple access permissions for a given reconfigurable object.

Assignments are method calls on the context overwriting the previous state. By convention, an user process must hold an access permission to invoke methods on the reconfigurable object. User processes requesting access permission for assignments additionally specify the runtime class for which a state is going to be written. The reconfigurable object automatically skips the reconfiguration step, if the current class matches the requested one. Even if there is no class switch, the constructor code is executed, as soon as a specific runtime class was requested. As a consequence, the user process can rely on the correct class being available, once it has been granted access. Additionally, the correct context is enabled. Third, the context's state is reset to that of a default-constructed object in case of an assignment. It is a common case for assignments to contexts to have default-constructed object values as source. In such cases the user process does not need to copy an object state into the context by means of another method invocation.

There is a subtle issue when using the anonymous access object. By definition, it does not determine which context is to be accessed. Therefore it may happen that the context enabled at the time of issuing a request differs from the context enabled at the time of being granted permission! There are two ways to cope with this situation. First, the management infrastructure may monitor the identity of the enabled context at the time of request. This would require extra hardware overhead. Also, to keep the context a valid target for the anonymous access, its lifetime must be guaranteed until the request can be serviced. This cannot be done for temporary contexts.

Therefore, the proposed approach uses a second solution: The anonymous access object may be seen as the context being enabled at the instant the scheduler grants access to a user process (and not at the instant the request is first seen by the reconfigurable object). This does not require additional logic and is always legal. Application code relying on a specific context to be enabled while not specifying its name is considered to be vulnerable to race conditions anyway and, therefore, disadvised.

Once a scheduler is required, a default algorithm is used. It is a fair scheduler, based

[2]This will be relaxed later in this chapter.

on a round robin strategy. The following pseudo-code sequence shows its `schedule()` method:

```
1  // "last_grant" remembers history
2
3  request_index schedule()
4  {
5    request_index next_grant = last_grant;
6    for ( request_index i = 0;
7          i < number_of_user_processes;
8          ++i )
9    {
10     next_grant = (next_grant + 1)
11                % number_of_user_processes;
12     if ( user process with id "next_grant"
13          has valid request )
14     {
15       break;
16     }
17   }
18   last_grant = next_grant;
19   return next_grant;
20 }
```

If this simple and area-saving algorithm is not suitable, a different one may be chosen. The algorithm needs to fulfill two requirements:

1. It must return the request index (which is a user process identification), of a valid request.
2. It must terminate in a finite number of clock cycles.

It can rely on:

1. The request table is updated to reflect the current situation.
2. The request table's values remain stable until the scheduling decision is made.
3. At least one will be valid.
4. The scheduler's state is unchanged since the last call. In the round robin example, this is the value of `last_grant`.

4.3.4 Reconfiguration Times

In order to reflect the expected system timing in the abstract model, the user needs to be able to specify the duration of class and object switches. An object switch from a permanent context to any other context includes saving the context state. Additionally, if the switch is done towards another permanent context, this may include a context restauration, too. It is assumed that the time required for saving or restoring a context is dependent on its runtime type. The reason being that the state size of different runtime types is likely to vary.

The time required for performing the partial reconfiguration is class dependent, too. Different logic implementations mapped to a specific reconfigurable area show a different logic utilization. This in turn leads to different logic bitstream sizes and different reconfiguration times.

Although the target architecture may support position independent bitstreams, this is not the normal case. Bitstreams are normally generated for a specific restricted area

on the FPGA device. It is likely that this already causes different bitstream sizes for any technology platform. Differences in routing to the static environment depending on the position of the reconfigurable area and different shapes of a reconfigurable area, even support the assumption of position dependent bitstream sizes. Since, these detailed bitstream size variations are not known early in the design cycle by principle, there is also a way to specify reconfiguration times independent of the position, indicated by the reconfigurable object.

Some architectures and their associated back-end tool-chains support *differential* bitstreams. Such bitstreams can be used, if the previous configuration is known. Differential bitstreams include only the differences between the old and new configuration. Dependent on the fraction of common elements of the old and new configuration, such bitstreams can be considerably smaller than regular partial bitstreams. This allows to speed up configuration times. To support differential bitstreams, the proposed approach supports specifying the previous configuration in terms of a source run-time class type. Differential bitstreams are preferred over regular partial bitstreams, if available.

Finally, the user might want to explore different FPGA devices with different reconfiguration speeds. To support this, device types are to be specified for which the timing declaration holds. By switching FPGA device types, the simulation can be easily adapted to different platforms.

Table 4.3 summarizes the possible elements of a timing annotation.

Entry	Optional	Explanation
Target class	No	Target class type indicating the required configuration.
Source class	Yes	Source class type of differential bitstreams. Not used in case of non-differential bitstreams.
Copy time	No	Attribute copy time for context state saving and restauration. This information is used when switching from or to permanent contexts.
Configuration time	No	Logic reconfiguration time for applying the bitstream. This is the duration of the specific class switch.
Reconfigurable object	Yes	Reconfigurable object instance. It specifies the partial area of which this annotation applies to. Can be omitted in early phases of the design cycle.
Platform	No	Specifies the FPGA device type. It is used to separate platforms with different reconfiguration speeds and bitstream sizes.

Table 4.3: Elements of timing annotations

4.3.5 Reconfiguration Scheduling

As described in Section 4.3.4, timing annotations are given for each object switch and class switch. However, this is not the only delay from the user processes' perspective. The user processes request access permission, demanding possibly a certain class type. The reconfigurable object schedules this request and executes it. Meanwhile, the user process is blocked.

Scheduling an access is required due to shared use of a reconfigurable object. For each reconfigurable object, there is a set of contexts bound to it. All accesses to contexts in this set are executed on the same hardware resource. This is resolved by the *access scheduler*.

There may be multiple reconfigurable objects in a system. Once at least two of these are mapped to a single hardware device, a second level of scheduling is required. The access controllers may independently decide to perform a FPGA context reconfiguration. Since, FPGA devices usually provide a single reconfiguration port only, these reconfigurations cannot be performed concurrently. Reconfiguration requests are transformed into a sequence during runtime to resolve this. This is the task of the *reconfiguration scheduler*.

Like the access scheduler, the reconfiguration scheduler has access to an information table about all requests. It is shown in Table 4.4. Like the access scheduler, the

Entry	Explanation
Valid-Flag	Flag, indicating if the request is valid. If false, all other entries are undefined.
Location	Target slot number.
New class	New class id for the reconfigurable area.
Old class	Old class id for the reconfigurable area.
Configuration cycles	Number of clock cycles required for reconfiguration.

Table 4.4: Request information table available to the reconfiguration scheduler

reconfiguration scheduler may maintain additional history tables. This allows sophisticated, history-aware schedulers.

So, the real costs in terms of runtime for a method invocation includes:

- Communication overhead between user process and access controller due to handshaking cycles.
- Computational time of the access scheduler, if other user processes are connected to the same reconfigurable object.
- Delays due to foreign processes, in case other requests are preferred by the access scheduler.
- Delay due to context saving, if needed.
- In case of a class switch:
  - Communication overhead between access controller and reconfiguration controller.
  - Computational time of the reconfiguration scheduler.
  - Delays, in case other requests are preferred by the reconfiguration scheduler.
  - Reconfiguration time.
- Delay due to context restauration, if needed.

Due to the interference of several user processes, schedulers and controllers it is difficult to quantify the delays using a fixed schedule.

On the other hand, this two-level scheduling avoids some interference between reconfigurable objects. As long as no class switches are requested, all reconfigurable objects operate completely independent from each other. Having one central instance managing accesses would likely introduce a central bottleneck. However, managing the reconfiguration port as a critical resource on a second level is inevitable.

To maximizse the time of independent operation of reconfigurable objects, the access scheduling algorithm may even depend on the current state of the reconfiguration controller. In Section 4.3.3, the list of parameters for each request is accompanied by a boolean status flag indicating the busy-state of the reconfiguration controller. This way, an access scheduling algorithm may delay requests requiring class switches, as long as a reconfiguration is being performed. This potentially increases application performance.

4.4 Optimization Techniques

4.4.1 Functional Density

A reconfigurable implementation can be compared to a static one by calculating the *functional density* as defined in [138].

> **Definition 23** - ***Functional density:***
> *The functional density D of a reconfigurable system is defined by $D = \frac{1}{AT}$ with A being the area and T the time to execute a fixed set of computations.*

The area A is unitless and represents the resources used for computation. For FPGAs, flip-flops, look-up tables, and routing resources could be used to quantify A. Other chip types need to be treated different, for example die area for ASICs. Functional density can be used to compare implementations targeted to the same FPGA. It is necessary to compare systems for which the same measurement of A can be used. This even makes comparisons between two FPGA families questionable.

In [138], functional density is used to decide, whether partial reconfiguration offers an improvement over a static implementation or not. A can be lowered by using dynamic partial reconfiguration (DPR). And since DPR introduces overhead, T may rise.

A is a measurement for FPGA resources. The author does not account for extra hardware in a more capable bitstream storage. The sum of partial bitstreams sizes is likely to exceed the size of a single full-device bitstream in case of a static implementation.

To be precise, A needs to account for both FPGA resources and bitstream storage resources (e.g. flash memory size). In this work, A is defined as $A = A_{FPGA} + \alpha * A_{storage}$. A_{FPGA} represents the logic resources on the FPGA while $A_{storage}$ reflects the area cost of the bitstream storage (flash memory, EEPROM or similar). A weighting factor α allows a trade-off between these two kinds of resources.

The time T may represent latency or bandwidth. The considerations in the following sections treat T as a measurement of bandwidth, even if both views can be justified. For a static implementation, T is the execution time $T_{execute}$ only. In the dynamic case, T needs to be split up. The approach proposed in this work allows for several reconfigurable areas operating concurrently. While inspecting a runtime trace for a reconfigurable area, one can find these modes: executing, idle, reconfiguring, saving, and restoring. These are shown in Figure 4.4. Additionally to this the communication overhead needs to be included.

The following definition for T is used in the proposed approach:
$T_r = T_{execute,r} + T_{idle,r} + T_{save,r} + T_{configure,r} + T_{restore,r} + T_{access\_schedule,r} + T_{config\_schedule,r} + T_{block,r} + T_{communication,r}$
With $r \in \mathcal{R}$ being the set of reconfigurable instances. The individual addends of the formula are:

T is the runtime for a given benchmark. All reconfigurable instances are in any of the given modes at any instant during runtime: $\forall r \in \mathcal{R} : T_r = T$. Therefore, it is not important, which reconfigurable instance to trace in order to obtain T.

When doing optimizations to improve D, the goal is to minimize the product of T and A. The use of reconfigurable resources can be tightly integrated in the application. Any resource (reconfigurable or not) may be idle due to data dependencies. As a consequence, it is not sufficient to seek the instance with the shortest idle time and optimize its use. However, it may be a good starting point.

In the following, a number of optimizations are introduced, which offer influence on the different addends to T_r and may also influence A. It is still the designer's task

Addend	Explanation
$T_{execute}$	Computation time
T_{idle}	Time without valid requests from user-defined processes
T_{save}	Time to save a context's dynamic state
$T_{configure}$	Time to reconfigure the FPGA
$T_{restore}$	Time to restore a context's dynamic state
$T_{access\_schedule}$	Time to calculate a new access schedule
$T_{config\_schedule}$	Time to calculate a new reconfiguration schedule
T_{block}	Blocking time due to a busy reconfiguration controller
$T_{communication}$	Handshaking overhead (user-defined processes, controllers)

Table 4.5: Summands for time in functional density

to find a suitable optimization for a specific application. Additionally, inspecting the composition of T_r gives advice which optimizations to try. These optimizations may influence other reconfigurable objects. Therefore, it needs to be verified after applying the optimization, if T is upgraded or degraded.

4.4.2 Trashing

If multiple user-defined processes employ a reconfigurable object, the timely utilization of the reconfigurable area is likely to be improved, that means, T_{idle} is reduced. The backside of multiple processors making frequent use of the reconfigurable object is, that it often needs to adapt itself to the various demands. Class switches potentially increase $T_{configure}$, T_{block}, $T_{config\_schedule}$, and $T_{communication}$. Object switches potentially increase T_{save} and $T_{restore}$ and may also include class switches. If the delays due to object switches and class switches represent a significant fraction of runtime, the *trashing* effect can be observed. Trashing is better known from processor caches, where several memory fractions are mapped to the same cache lines. Due to an unwise memory layout and adverse order of memory accesses very frequent displacement effects occur. As a result of frequent cache misses, the application performance is decreased.

Figure 4.6 illustrates two trashing situations. In this example, two permanent contexts are used in rotation by two processes. A vertical bar represents the reconfigurable object. The code executed by the user-defined processes is sketched left and right of each vertical bar. The left process uses c_0 and the right one uses c_1. The height of the source code statement indicates a method's execution.

It is easy to see that for each execution at least a state-saving step and a state-restauration step is required. The situation on the left is showing the case of c_0 and c_1 having the same dynamic class types at runtime, whereas the right side shows different runtime types. In the later situation, a class switch is required for each object switch, resulting in a reconfiguration.

This is acceptable, if the systems troughput is secondary (which is rarely the case), or if the switch times are negligible compared to the computation time for each method invocation. Since, such situations are unusual, the reconfigurable object's use needs to be optimized.

For optimal troughput, a situation as depicted in 4.7 is preferred. It shows the same situation without interleaved accesses to the contexts. While it clearly avoids overhead times, it might be dangerous to forbid interleaving. Executing methods on c_0 two times before c_1 is enabled, increases the blocking time of the user process waiting on access permission to c_1. Finding a compromise between latency and throughput of a given application, while considering other demands like fairness and starvation freedom is a general scheduling problem. Both access scheduler and reconfiguration scheduler must find a suitable sequence at runtime. To maintain a short development cycle, the

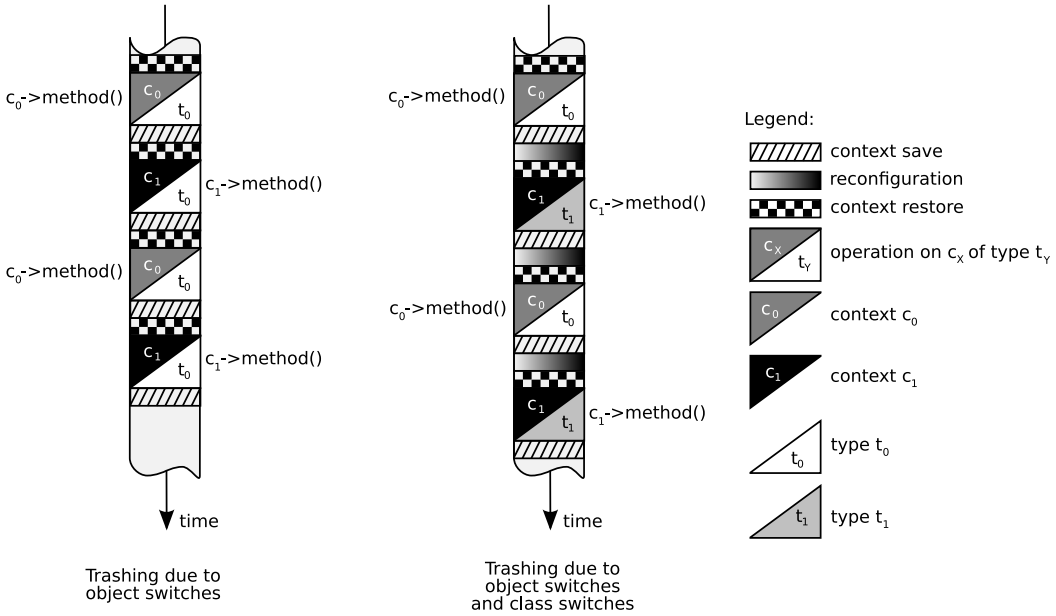

Figure 4.6: Execution with trashing effects

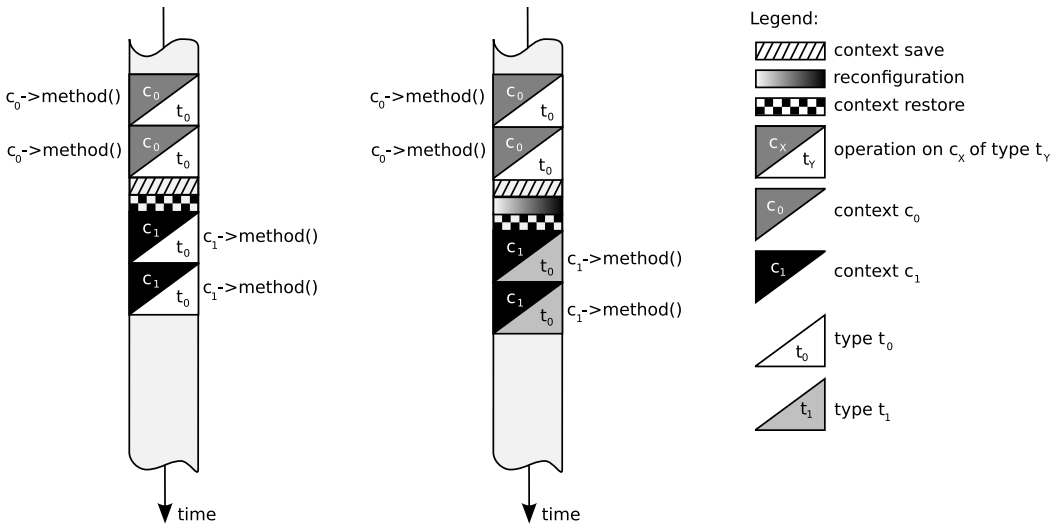

Figure 4.7: Execution without trashing

designer would want to stick to well-known and pre-defined scheduling algorithms. Unfortunately, in some cases the algorithm needs to be tailor-made for the application. User-defined schedulers are possible in the presented approach. These schedulers would then be interwoven with the application code. This is not always the best way to cope with such a situation.

4.4.3 Locks

Example: Cryptography unit. Consider a network stream processing unit, which encrypts and decrypts datastreams as a service for other components. Let u_0 and u_1 be two user-defined processes and r a reconfigurable object as shown in the simplified block diagram 4.8. Two permanent contexts, c_0 and c_1 are bound to r. u_0 uses c_0

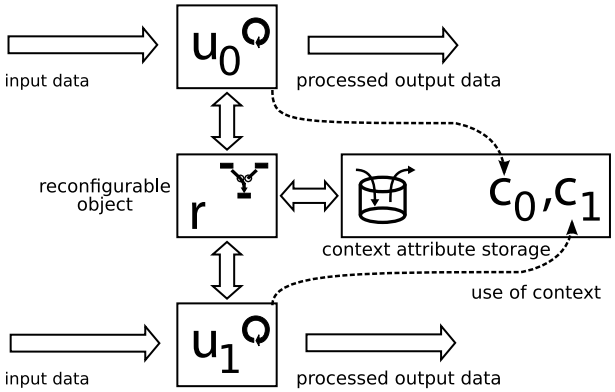

Figure 4.8: Simplified block diagram for cryptography example

and u_1 uses c_1 exclusively. Both permanent contexts are of a class type implementing cryptographic operations. Among other algorithms, the set of algorithms includes a

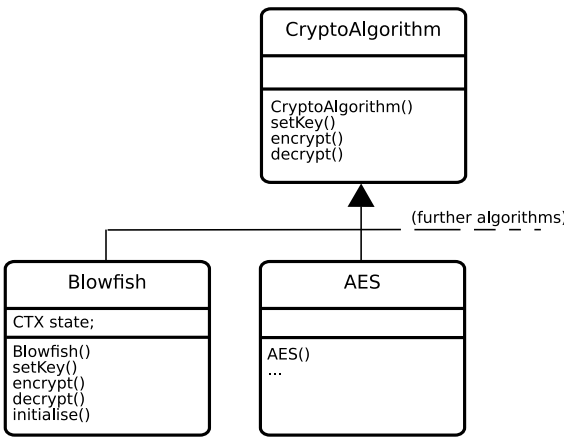

Figure 4.9: Class diagram for cryptography classes

Blowfish[118, 43] implementation. A class tree is sketched in Figure 4.9. Blowfish needs to be initialized with the encryption keys. The initialization sequence fills an internal datastructure (`ctx`), which can then be used to perform encryption and decryption. So, Blowfish can be used with this sequence:

1. Set cryptography keys

2. Initialize, build `ctx`
3. Encrypt or decrypt data

Scenario A: Usage-dependent optimization. If u_0 detects need for a specific algorithm it changes c_0's dynamic type to the appropriate class. Next, it follows the sequence of setting keys, initializing and finally encrypting or decrypting the first chunk of data. To optimize performance, the designer decides to forbid disabling the context until the sequence terminates by processing the first chunk of data.

The characteristic property of scenario A is the required information to optimize the schedule being located in the user-defined process (and not inside the scheduler). To express such a constraint, *external locks* and *lazy permission handling* are used.

Definition 24 - ***External lock**:*
An external lock is a Boolean flag indicating whether an enabled permanent context may be disabled. These flags are modified from the outside of the reconfigurable object by the user process.

There is one external lock for each context and user process. The flag defaults to `false`, allowing a context to be disabled. If the user process does not contain manipulation statements for an external lock, it is omitted and replaced by the default. Setting this lock does not *enable* a context, it just controls whether it may be *disabled*, once it becomes enabled. External locks can be used to prevent object switches in performance critical sections inside the user process. They do not provide exclusive access to a context. Other processes may continue to manipulate the same context. This is necessary, whenever the context also plays a role as a inter-process communication object.

Scenario B: Exclusive access. Without a communication role, the shared manipulation of a context may be unwanted. Then, an exclusive use of a permanent context is preferred. The user-processes can achieve this by keeping the access permission over several method invocations. Figure 4.5 depicts the state diagram of the user processes regarding their interaction with reconfigurable objects. Keeping the access permission over several method invocations refers to the edge from *Execute* state to itself.

There is research showing that such an optimization can have a very positive influence on the system's performance [102]. As expected, this optimization needs to be applied carefully.

In the presented approach there are two modes for user-defined processes. These are managed independently for each reconfigurable object and control the user-defined processes' policy for returning access permissions.

While in *swift* (which is the default) mode, the access permission is returned after each access, in order to maximize the scheduler's freedom. In *lazy permission* mode, the process keeps the permission until the mode is left or a different context is to be accessed.

Definition 25 - ***Swift permission mode**:*
The user-defined process returns access permissions to the reconfigurable object after each method invocation.

Definition 26 - ***Lazy permission mode**:*
The user-defined process returns access permission on object switches or when entering swift mode.

Upon each context access to be performed by an user-defined process the process checks whether it holds an old permission. If the old permission is for a different context bound to the same reconfigurable object, it is returned first. If it is for the same context, the old access permission is re-used.

When performing anonymous accesses in lazy permission mode, the user-defined process does not know which context (permanent or temporary) the permission was granted for. This is not an issue for a sequence of anonymous accesses. If an anonymous is followed by a named access, it is unclear to the user-defined process whether the required context matches the enabled one. Therefore the user-defined process acts conservatively: It assumes an object-switch happens and returns the access permission for the anonymous access object.

Scenario C: State-dependent optimization. In some situations the context itself is the best component to decide whether to allow disabling itself or not. In case of the cryptography example, the context's dynamic state may tell if the initialization was performed after setting new keys or if the first encoding or decoding was performed after the initialization.

In such cases, internal locks are suitable:

> **Definition 27** - ***Internal lock***:
> *An internal lock is a Boolean equation implemented as a member function of a class. It operates on the classes' data members. An enabled context may not be disabled, as long as the expression evaluates to* `true`.

Like external locks, the internal counterparts do not cause a context to become enabled. Internal locks allow specifying some invariants for disabled contexts. Since the elements of the expression are the context classes' data members only, the internal lock solely depends on the dynamic state. These can be specified without taking the user defined processes into account. Internal locks are defined for both temporary and permanent contexts.

Enhanced Access Control

The access control built into reconfigurable objects takes internal and external locks into account. A user-defined processes' request is treated as temporarily not serviceable, if servicing it would require to disable a locked context. Such requests are not marked as valid in the access scheduler's request table. As a consequence, they remain hidden until the internal lock and all external locks are released. By construction, the locking mechanism can be used with any kind of access scheduler. In other words, the access scheduler doesn't need to be modified to support it.

The designer should apply optimizations depending on where the information required for well-grounded decisions reside. Table 4.6 gives an overview.

Location	Kind of Information	Optimization
User-defined process	Process' state of control	External lock or lazy permission mode
Context	Attribute's values	Internal lock
Scheduler	Resource history	Custom scheduling algorithm

Table 4.6: Optimization techniques

For some applications, clever use of locking and lazy modes may improve the functional density. On the other hand, overly excessive locking may also reduce system performance. Both effects were demonstrated in [102].

4.4.4 Transient Attributes

Due to the definition of permanent contexts, their dynamic state may never get lost unless a system reset occurs. Method invocations on permanent contexts may modify the context's state but the state must not be influenced by operations on different permanent contexts. Functional interference between contexts would immediately break the concept of virtual hardware.

For this reason, switching permanent contexts requires saving the previous context's state and restoring the upcoming context's state. A context attribute storage component preserves the state of all disabled contexts. The memory reserved for a permanent context's data may be considerably large. This is not uncommon or artificial. As an example, the Blowfish algorithm requires 1024 bytes of memory for the CTX and up to 21 bytes for the encryption key.

To save some area costs of the context attribute storage and to shorten the copy times, a further optimization can be made. Data attributes in the context's runtime class type may be statically flagged as being *transient*.

Definition 28 - ***Transient attribute**:*
A transient attribute is a flagged data attribute of a context's runtime class type. Transient attributes are not saved upon disabling a context. When enabling the context, they are reset to default values. Transient attributes behave as regular class attributes if used outside a context.

The concept of transient attributes is known from software programming languages like Java. Java distinguishes between copy operation and serialization [126]. Copy operation preserves transient attributes while serialization does not. Normal copy operations in Java are *shallow*: References to other objects are copied instead of copying the referenced objects themselves. Object serialization is the process of transforming an object's state into a sequential datastream[36]. Serialization implements a *deep* copy: Referenced objects are copied, too. Serialization creates new instances of these attributes. Sometimes a deep copy is desired, expecially in network programming or object databases. The straight forward way to implement a deep copy in Java is to add a suitable method to each class and recursively copy whole datastructures (e.g. graph structures). A different solution would be to use serialization for deep copy. While this is possible, it introduces some pitfalls, since it clears transient attributes. The application's class tree needs to be carefully hardened to handle this situation.

When handling contexts, similar problems occur. To save a context's state, the object is serialized and stored in the memory. Restoring it includes deserialisation. During serialization, transient attributes are omitted. Deserialisation restores transient attributes with predefined values. Enabling one context might require disabling another one. This in turn would clear the transient attributes of the previous one. From other user-defined processes' view, the transient attributes would be spontaneously reset to their default values without any visible reason. Obviously, this would render transient attributes unreliable and therefore, unusable to store information.

In case of the cryptography example, the designer might introduce a boolean attribute, indicating whether the CTX is valid or not. Prior to encoding or decoding data, the flag could be checked and the initialization can be performed on demand. Setting a key resets this flag to `false`. It is marked as transient with `false` being the default

value, too. Initialization sets the flag to `true`. This would cause a re-initialization each time a context was disabled.

Transient attributes may reduce T_{save} and $T_{restore}$ in the functional density calculation. Additionally, A is reduced due to a smaller storage for context states.

A combination of transient attributes and internal locks is even more powerful. By temporarily preventing contexts to be disabled, the transient attributes become reliable again. For example, the internal lock could be set to `false` (lock being inactive), once a required number of encoding or decoding operations was performed: `internal_lock = number_of_ops < 20` or similar. In other applications, internal locks may protect transient local buffers from loosing their state unless they are empty. The right combination of transient attributes and internal locks is heavily dependent on the application.

Transient attribute markers are meaningless for temporary contexts, since those are never being serialized.

4.4.5 Slots Inside Reconfigurable Areas

By using locking mechanisms and lazy mode in user processes, the scheduler is restricted in its decisions. While this reduces object switches, it may have a negative effect on the functional density due to increased blocking times for user processes. This leads to increased T_{idle} for other reconfigurable objects.

A solution would be to balance the use of reconfigurable resources, minimizing idle times. This needs to be done very carefully, since it might introduce trashing in other reconfigurable object's traces.

Contexts would need to be migrated from one reconfigurable object to another to implement load balancing. Manipulating a context would then start with a localization step to find out, *where* a context is currently migrated to. A central broker instance would be required to localize the context and to decide about migrations. Such a central instance is likely to become a bottleneck. Another drawback would be, that any user process would need an interface to any reconfigurable object that is a potential host for the desired context. Especially in routing-limited devices, such as FPGAs it is advisable to limit the number of possible connections. This rules out a point-to-point connection style. A bus-style would introduce another bottleneck.

A suitable solution to such a dilemma often is a compromise. The model of isolated resources as introduced in Section 2.3 is too restricted to allow balancing. In the other extreme, the model of one shared resource pool introduces bottlenecks like brokers and busses. Plus, automated analysis of access dependencies is in-between extremely difficult and impossible.

In the proposed approach, the model of isolated resources is softened to a model of isolated groups of resources. Figure 4.10 extends Figure 2.3 by introducing the softened model. For each reconfigurable object, the designer may split the reconfigurable area into a set of *slots*[3]. Each slot may be used to enable any context that is bound to the reconfigurable object.

By doing so, the reconfigurable object acts as an object broker. The designer may tune the system to its needs: One extreme implementation strategy would be to bind all contexts to a single reconfigurable object. This would resemble the shared resource pool model. The opposite strategy would be to use isolated resource pools, having a group size of 1 per reconfigurable object. Finally, the designer may explore the design space by modifying the bindings of permanent contexts to reconfigurable objects and adjusting the slot set size of reconfigurable objects.

The proposed approach restricts reconfigurable objects with multiple slots to named accesses. Anonymous accesses are ambiguous if the reconfigurable object has multiple slots. As a consequence, no temporary context is available for such reconfigurable

[3]The slots may even be physically separate, depending on the FPGA vendor's back-end tools.

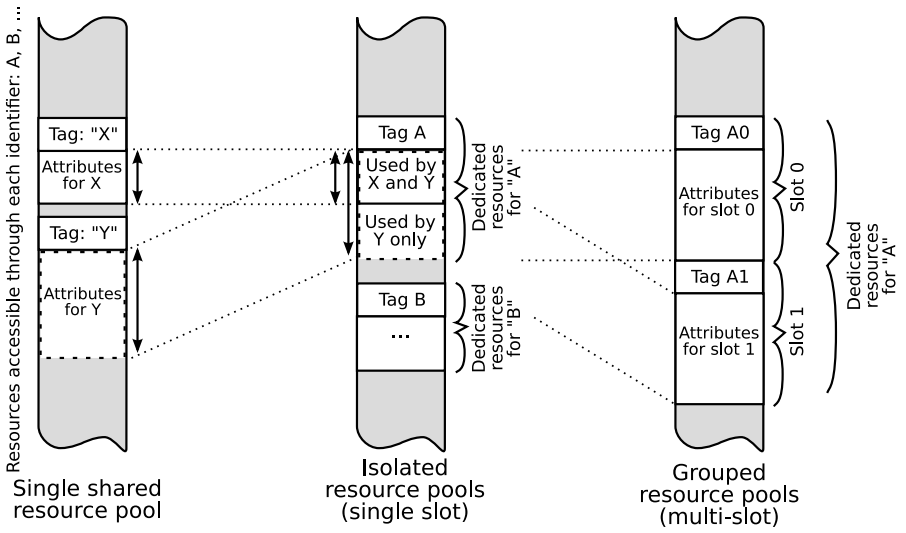

Figure 4.10: Resource groups

objects, since it requires anonymous assignments. Overcoming this obstacle means more than just finding convenient ways to specify slot identifications. It would also require ways to manually locate contexts, introduce locate-and-lock mechanisms, etc. It is likely that such systems are better built using single-slot reconfigurable objects and manual context migration.

Communication Network

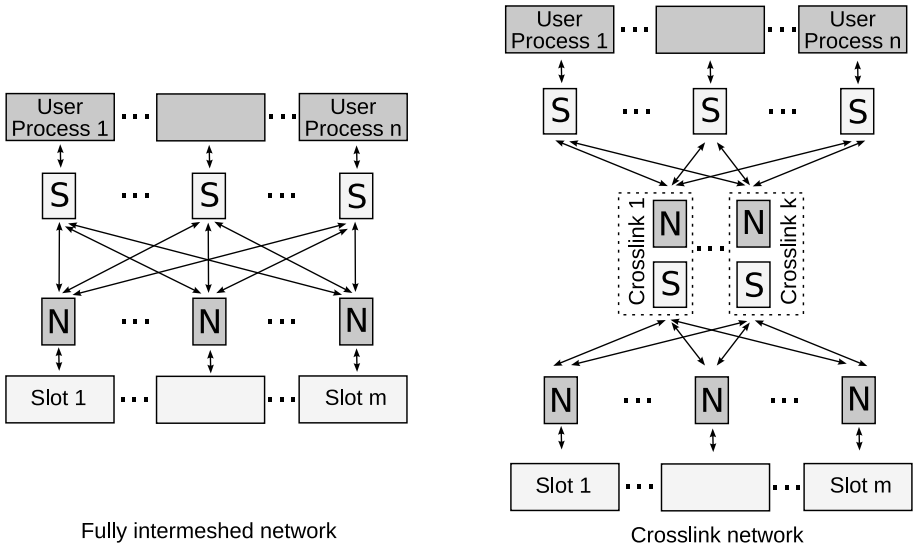

Figure 4.11: Abstract crosslink block diagram

While using resource groups of adjustable size for reconfigurable objects, there is still potential to minimize routing resources. Consider a large number of user-defined processes communicating with a multi-slot reconfigurable object. In this case, the number of point-to-point connections may become unacceptably large. Figure 4.11 depicts

an alternative offered by the proposed approach. There are n user-defined processes communicating with the object and m slots. In a full-intermeshed network as shown on the left side, $n * m$ connections need to be implemented. When restricting the number of concurrent accesses to k, one crosslink for each connection is implemented. The crosslink variant requires $n * k + k * m = k * (n + m)$ connections. Especially when k is small, routing resources can be saved. A crosslink implementation with a small number of links resembles processor caches: Multiple contexts may be in enabled state simultaneously, just like a cache n-way associative processor cache (with $n > 1$).

Context Placement

To enable contexts in a reconfigurable object with multiple slots, a placement decision has to be made. The outcome of this decision influences the costs to service a user-defined processes' request.

There have been many works on temporal and spatial placement strategies for reconfigurable systems (e.g. [46, 88, 136, 10, 50, 123, 41, 105, 77, 34]). Especially those strategies, which generate placement decisions for slot-based systems may be adapted to manage multi-slot reconfigurable objects. The proposed approach does not include a proposal for a specific algorithm but tries to make implementations of known algorithms an easy task.

The following information is provided for each of the slots:

Entry	Explanation
Allocated flag	Indicating if it contains a valid permanent context.
Context in slot	Identification of the context currently occupying the slot.
Class	Type identification of the current class or $\bar{t}$ if invalid.
Granted flag	Indicating if a grant was given to a process.
Free flag	Indicating if this slot may be used for different contexts.

Table 4.7: Slot information available to placement algorithm

The entry is marked as not free, if any user process currently holds an access grant to the slot or if the slot contains a locked context. In these cases, context field indicates the only context which may be placed here. The "free" flag is not stored permanently but re-calculated for each round of placement decisions. The other fields are stored permanently and updated on demand.

The context to be placed is described as shown in table 4.8: All entries are stored

Entry	Explanation
Enabled flag	Indicating whether the context is currently enabled.
Slot identification	If enabled, this yields the slot number.
Lock flag	Indicating if the context is currently locked.
Class identification	Identification of the context's current runtime type.

Table 4.8: Context information available to placement algorithm

permanently and updated on change only with the locking information being the only exception. The locking flag is calculated by *or*-reducing all external locks and the internal lock. Additionally to these two sources of information, the placement algorithm may maintain additional information tables in order to be history-aware.

The placement algorithm is called on each context to be placed. Afterwards, the scheduler is invoked. Timing costs presented to the scheduler reflect the impact of the placement algorithm's decision.

For example, consider a placement algorithm being able to re-use configurations to a great extent. In this case, the costs to service a request that are presented to the scheduling algorithm are lower. A cost-sensitive scheduling algorithm may then pick the requests, which are cheap to service first.

Servicing a request may include object switches and class switches. Such switches lead to a modified slot description as shown in table 4.7. As a consequence, the table is re-calculated every time a new scheduling decision needs to be made. It is possible that the placement algorithm does not find a slot for the context to become enabled. In this case, the placer may return a special value $\overline{s}$, indicating that this request cannot be served at the moment. Such requests are presented as not valid to the scheduler.

The default algorithm is to re-use slots with matching configurations first. Its pseudo-code is shown below, using `no_valid_slot_found` for $\overline{s}$.

```
1  slot_index place(context_descriptor upcoming_context)
2  {
3    // 1st pass, search for a slot that is
4    // preconfigured with the desired class
5    foreach (slot_index in 0 to number_of_slots-1)
6    {
7      // Slot free and preconfigured with the desired class?
8      if ((slots[slot_index].free) &&
9          (slots[slot_index].class == upcoming_context.class)
10       )
11     {
12       return slot_index; // Found! Abort search.
13     }
14   }
15   // 2nd pass, only if no ideal slot was found
16   foreach (slot_index in 0 to number_of_slots-1)
17   {
18     // Don't care for matching class, "free" is enough
19     if (slots[slot_index].free){  return slot_index;  }
20   } // for
21
22   return no_valid_slot_found;
23 }
```

Summary of Optimizations

A summary of expected influences of the various optimization strategies can be found in Table 4.9. Idle times may be increased by locking mechanisms and lazy permission mode, since they may increase blocking times. This may lead to unbalanced use of reconfigurable areas. However, all three mechanisms may greatly limit trashing effects, reducing overhead due to overly frequent object switches and class switches.

Transient attributes shorten saving and loading dynamic states. As a second order effect, using these may also have a positive influence on blocking times, load balancing and therefore, idle times. Note, this table does not show the positive effect due to a reduced context storage size.

Having multiple slots controlled by a reduced number of reconfigurable objects is likely to demand more time for access scheduling. Furthermore, it increases resource demands to implement communication due to the increased flexibility. This cannot be seen in the table. The advantages are those of caching effects and a better balance in utilization. This reduces object switch and class switch frequencies and also reduces blocking times.

Influence on	Optimizations				
	Locks		Lazy permission	Transient	Multi-slot
	External	Internal	return mode	attributes	areas
$T_{execute}$					
T_{idle}	-	-	-	+	++
T_{save}	++	++	++	++	++
$T_{configure}$	++	++	++		++
$T_{restore}$	++	++	++	++	++
$T_{access\_schedule}$	+	+	+		-
$T_{config\_schedule}$	+	+	+		
T_{block}	--	--	--	+	++
$T_{communication}$					

Table 4.9: Summands for time in functional density

As it can be seen from the table, the proposed optimization strategies provide a toolbox. The individual strategies need to be applied as demanded by the specific application. It is not possible to give a general advice, which ones to use and which to avoid.

4.4.6 Refined Context Management

Having introduced internal and external locks, transient attributes, multi-slot reconfigurable areas, and placement algorithms, a refined algorithm for servicing requests in reconfigurable objects can be specified. The following sequence is started whenever the access control component is in idle state and at least one user-defined process indicates a request.

1. Collect requests:
   (a) Mark all requests as valid (serviceable). All requests will be presented to the scheduler, unless a reason for denial is found. Until then, they are all treated as serviceable.
   (b) User-defined processes may ask for the following actions: requesting and returning permissions, creation of objects, switching contexts, and locking or unlocking of contexts. Only permission requests, context switches or creation of contexts may collide. All other actions are served independently of all processes in a concurrent manner. For permission grants, a maximum number may be specified. Later on, this helps limiting the amount of resources required to connect the user-defined processes with the reconfigurable areas. If the maximum number of grants for this reconfigurable object is reached, flag all requests for new permissions as non-serviceable. Requests for context creation are not affected. Context switch requests are always combined with permission requests in case of multiple user-defined processes connected to the reconfigurable object. They then don't occur in isolation. In case of a single user-defined process, using a reconfigurable object exclusively, the maximum number of grants will always be one and never exceeded. So, context switch requests do not need to be treated in this step.
   (c) Resolve anonymous accesses to the currently enabled context. The user-defined process are unaware of the currently enabled contexts unless they hold an access permission for a permanent context. However, the request needs to be resolved to a real context. Non-assignment statements can be identified by not requesting object creation but only access permission. These are resolved to the currently enabled context by the access control

component. Anonymous assignments are always resolved to the temporary context (see Section 4.3.2). The resolution is not permanent for all subsequent iterations of this sequence. Since, it may take several scheduling rounds until an anonymous request is serviced, the currently enabled context might change. Therefore, the resolution is done for each scheduling round again.

(d) Calculate "lock" flags in the context descriptor table (see Table 4.8). This indicates, whether enabled contexts may be disabled or not.

(e) Calculate "free" flags in the slot descriptor table (see Table 4.7). These flags indicate whether a slot may take a new context or if it is occupied with another context, which is locked or for which permission grants were given.

(f) Place all contexts by picking targets for all disabled contexts. Mark all contexts that cannot be placed as non-serviceable.

(g) If no serviceable requests remain, end this sequence and start over in next clock-cycle.

(h) Calculate "save" and "load" flags, save and load cycle count, and configuration cycles in the request information table to be given to the scheduler (see Table 4.1). The other fields are fixed or directly read from the user processes' request.

2. Execute selected request:

(a) Execute scheduler and let it pick a request to be serviced.

(b) If currently enabled context is a permanent one and no class switch was requested, save the current' context's state. Omit transient attributes while doing this.

(c) If class switch is necessary, reconfigure the selected slot.

(d) If class switch was requested (necessary or not), reset slot.

(e) If the new context is a permanent one and no class switch was requested, load the context's state. Afterwards, set transient attributes to their default values.

(f) If access permission was requested, modify crossbar settings to connect user process and slot and then grant permission.

(g) Update the permanent fields of the slot descriptor table and context descriptor table.

This algorithm does not include setting and removing external locks. Just like returning a permission, these operations may be done independently of the algorithm as described above. An user process may indicate such a request and does not need to wait for acknowledgment.

4.4.7 Roundup: Elements of the Reconfigurable System

Figure 4.12 summarizes the elements of a reconfigurable system as proposed in this chapter. It consists of a set of user-defined processes (indicated by circular arrows), communicating with reconfigurable objects.

As a lookahead to the implementation, the contents of the reconfigurable objects are shown in the upper magnifier glass. Reconfigurable objects always contain an access scheduler and a reconfigurable area, separated into slots. In case of permanent contexts being bound to the reconfigurable objects, a context storage preserves the dynamic states of disabled contexts. If the reconfigurable area is split up into multiple slots or if multiple user-defined processes use the object, a crossbar is introduced to connect user-defined processes and slots. The crossbar is controlled by the access control component.

The reconfigurable objects decide about required reconfigurations. A reconfiguration control instance executes them. The lower magnification glass shows the contents

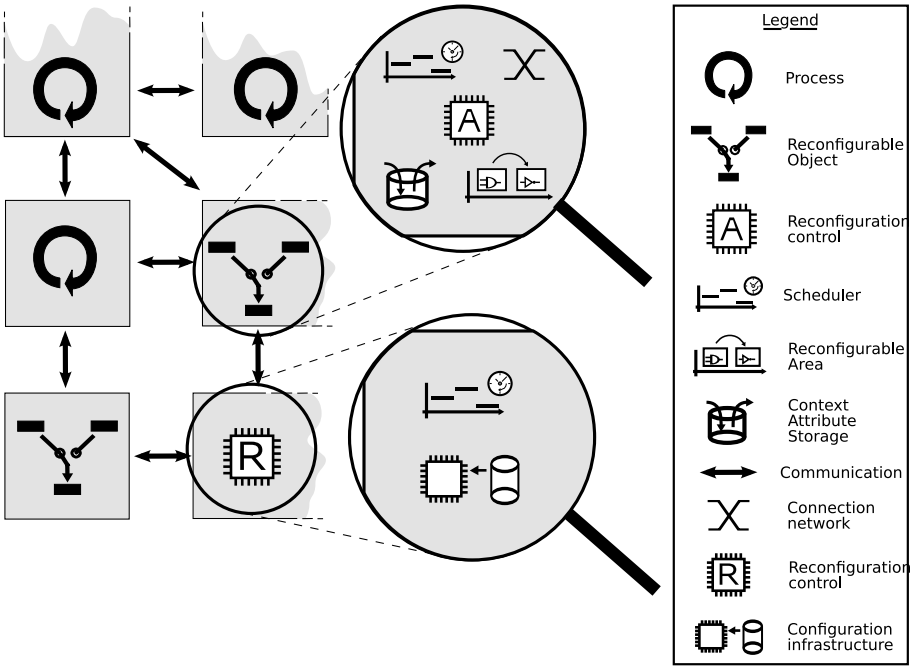

Figure 4.12: Reconfigurable objects and their content

of the reconfiguration control component. It contains a scheduler in case of multiple reconfigurable objects. It also contains the control logic to reconfigure the FPGA device.

An OSSS+R model has explicit instantiations of user-defined processes, reconfigurable objects and reconfiguration control components. However, their contents and their interconnection is derived from the used types, the connection topology and some optional parameterization statements. How this is embedded into an SystemC-based model is presented in the next chapter.

4.5 Chapter Summary

This chapter presented a modeling framework for object-oriented descriptions of reconfigurable hardware. Polymorphism as provided by C++ was analyzed and compared to the static hardware implementation in OSSS. Then, it was applied to partial runtime reconfiguration. Low-level control was described by introducing anonymous accesses and the temporary context. More convenient high-level abilities were demonstrated using named accesses and permanent contexts, allowing virtual hardware. A two-level scheduling with access management, as its first level and reconfiguration management as its second level was introduced. Using the functional density metric, different runtime optimization and area-saving strategies were presented.

Chapter 5

Simulation Semantics of OSSS+R Models

This chapter presents the simulation library for OSSS+R.

First, the mandatory elements to describe model structure are introduced. They are followed by a description of optional modifications that allow expressing optimizations as introduced in Section 4.4. Having presented the structural elements of a model, the simulation elements are shown. These describe operations at runtime, e.g. method invocations or assignments. Finally, the chapter is concluded with a set of possible extensions to the simulation library and a chapter summary.

5.1 Mandatory Structural Modeling Elements

When introducing partial reconfiguration, some changes to a model are obligatory. To ease the understanding of these changes, an artificial initial model (shown in Figure 5.1) is modified stepwise to match Figure 4.12 that concluded the previous Chapter 4. To highlight the differences to the previous step, new or modified parts are shown in black, while unchanged parts are shown in grey.

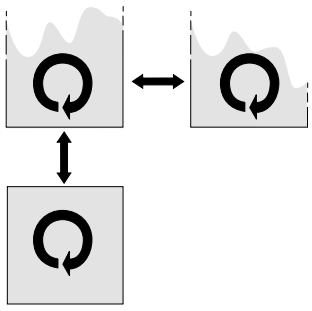

Figure 5.1: Artificial initial model

Introduction of Reconfigurable Objects

Reconfigurable objects are represented as instances of the `osss_recon<T>` template class. Since reconfigurable objects are entity types, they have to be placed as data members inside SystemC modules. A single argument `T` is expected for the template `osss_recon<T>`. T represents the static class type that is used to interface the object.

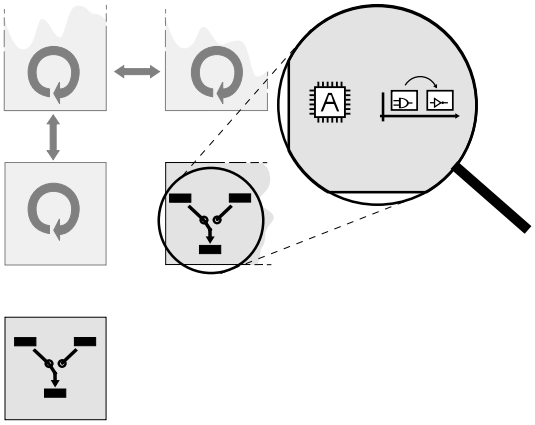

Figure 5.2: Model with reconfigurable objects

Reconfigurable objects have two port members of type `bool`, named `clock_port` and `reset_port`. They are to be bound in the module's constructor, treating the reconfigurable object as a sub-module. Reconfigurable objects have a single constructor, that is used the same way as those of SystemC modules or ports:

```
osss_recon( sc_core::sc_module_name =
              sc_core::sc_gen_unique_name("osss_recon") );
```

When called without an argument, a default identifier with the naming prefix string `osss_recon` is created. Otherwise, the passed string is taken as an identifier. Figure 5.2 shows the instantiation of reconfigurable objects. The magnifying glass shows the reconfigurable area, as well as the access control component embedded inside the reconfigurable object.

Declaring Use of Reconfigurable Objects

A reconfigurable object can be viewed as a service provider to user-defined processes. It provides the execution environment of contexts that can be manipulated by the user-defined processes, which act as clients. Since, all structural properties of the model must be determined at the elaboration stage, the reconfigurable object must get to know which clients to serve. This, for example, influences the presence of schedulers and their properties. To declare which user-defined processes make use of a reconfigurable object, the `uses()` or `osss_uses()` statements are provided. By utilizing `osss_uses()`, the designer makes it clear, that an `osss` keyword is used. Similar to this, most predefined SystemC keywords start with the prefix `sc`. There are exceptions to this rule, for example SystemC's `reset_signal_is()`. Since, `osss_uses()` is to be placed directly before or after `reset_signal_is()`, the short form `uses()` (without prefix) is provided as well. An example is:

```
SC_CTOR( myModule ) : my_recon_obj("my_recon_obj")
{
  SC_CTHREAD( myThread );
  reset_signal_is(rst_port, true);
  uses( my_recon_obj, 42);
  ...
}
```

An `uses()` statement refers to the least declared process, just as `reset_signal_is()` does. It states that the process contains manipulation statements of the reconfigurable

object passed as a first argument. The second argument is optional and may be omitted. It is a numeric value representing a priority value. The value is fixed and made available to the scheduling algorithm, if there is one. Figure 5.3 shows the effect graphically. If

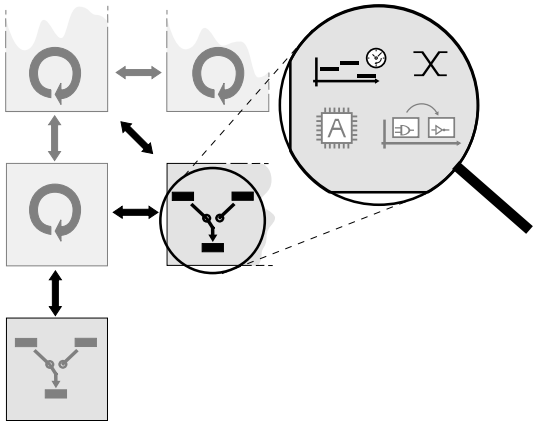

Figure 5.3: Model with usage statements

there are at least two user-defined processes interacting with the reconfigurable object, a scheduler is created. Also, a crossbar is inserted to implement mutual access to the reconfigurable area by the user-defined processes.

Introduction of Persistent Contexts

For the simulation phase, anonymous manipulations will be expressed using the identifier of the reconfigurable object itself. To allow manipulation of permanent contexts later on, there must be similar symbols to allow an identical syntax. Therefore, permanent context names need to be introduced. This is done using the `osss_context<>` keyword. Like reconfigurable objects, permanent contexts are to be instantiated as data members of modules. It is not necessary that they are embedded inside the same module as the reconfigurable object. They are bound to reconfigurable objects explicitly using the `bind()` function. These bindings are to be placed inside the `sc_main` function or in the top-level module's constructor. An example:

```
1  SC_MODULE( myTop )
2  {
3    ...
4    osss_recon< A > my_ro;
5    osss_context< B > my_c;
6    ...
7
8    SC_CTOR( myTop ) : my_ro("my_ro"),
9                       my_c("my_c")
10   {
11     ...
12     my_ro.bind(my_c);
13     // alternatively, more common in SystemC:
14     // my_ro( my_c );
15   }
16 }
```

Instead of the `bind()` method, a less verbose syntax more common in SystemC models is possible, using braces only. The construction of persistent context is the same

as those of the reconfigurable objects. The constructor accepts an optional string to name the object. Figure 5.4 shows the change graphically. Once permanent contexts

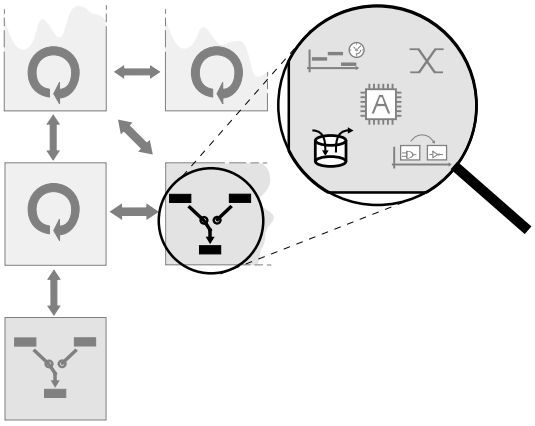

Figure 5.4: Model with permanent contexts

are bound to a reconfigurable object, a context attribute storage is added to allow the saving of the declared context's states.

Persistent context identifiers may also be used as arguments for `osss_uses` statements instead of reconfigurable objects. A process that is declared to use a persistent context is implicitly declared to use that reconfigurable object to which the persistent context is being bound. This simplifies modeling, since the designer may alter the binding of persistent contexts and reconfigurable objects to optimize a design. By declaring use of the persistent contexts instead of reconfigurable objects, the `osss_uses` statements need not to be modified, if the persistent context binding changes.

It is permitted that multiple `osss_uses` statements declare use of the same reconfigurable object by an user-defined process. Since superfluous statements are ignored, it is safe to add `osss_uses` statements for each persistent context used by a process.

Introduction of Device Types and Instances

Reconfigurable objects are being reconfigured at runtime but they don't reconfigure themselves. Instead, they detect the demand for doing so, but leave the reconfiguration process to be performed by another component. This reconfiguration control unit is instantiated once per device on the platform. The platform may consist of devices of different types. Additionally, the designer may want to explore the design space by switching between device types.

The device types differ in their reconfiguration properties. Depending on the logic to be configured, the simulation of the reconfiguration process has a different duration. To instantiate a device type, the following constructor may be used:

```
1 osss_device_type(const std::string name);
```

An example with a device type and a timing annotation:

```
1 osss_device_type virtex4("xcv4");
2
3 OSSS_DECLARE_TIME( virtex4,
4                    MyClassA,
5                    sc_time(12, SC_NS),
6                    sc_time(40, SC_MS) );
```

These two statements declare a device type with the identifier `virtex4` that has a name string `xcv4`. The name string is used for debugging, logging and may later on assist in the selection of platform support packages after synthesis. The second statement declares that configuring the user-defined class `MyClassA` lasts 40 milliseconds. Saving and restoring the state of instances of type `MyClassA` lasts 12 nanoseconds.

These statements are to be placed inside the SystemC `sc_main()` function or inside the constructor of the top-level module in the design hierarchy. The formal definition of a simple timing declaration is:

```
OSSS_DECLARE_TIME( device_type ,
                   target_type ,
                   attribute_copy_time ,
                   logic_reconfiguration_time )
```

There are further variants of this statement for special situations:

```
OSSS_DECLARE_SHORTCUT_TIME( device_type ,
                            target_type ,
                            source_type ,
                            attribute_copy_time ,
                            logic_reconfiguration_time )
```

This is a special variant for a reconfiguration shortcut. Some platforms support differential bitstreams. These allow shorter reconfiguration times, if the previous logic is known, since the reconfigurations can be restricted to their differences. This declaration has precedence over `OSSS_DECLARE_TIME`.

```
OSSS_DECLARE_SHORTCUT_TIME_FOR_RECON(
                     device_type ,
                     target_type ,
                     source_type ,
                     attribute_copy_time ,
                     logic_reconfiguration_time ,
                     recon_obj )
```

This is an even more specialized statement. It additionally allows specifying the reconfigurable object to which it applies. Since the reconfigurable areas for reconfigurable objects may have different shapes, the reconfiguration times may differ. Individual timing can be specified using this statement. It has precedence over `OSSS_DECLARE_TIME` and `OSSS_DECLARE_SHORTCUT_TIME`.

A fourth version, which also allows specifying the slot number for which an annotation is meant, would be required to be complete. This, however, was not implemented in the simulation library yet.

After device types have been defined, device instances may be created. This step is depicted in Figure 5.5.

In the model, this is done using this statement:

```
osss_device(
      osss_device_type & new_type,
      sc_core::sc_module_name module_name
        = sc_core::sc_gen_unique_name("osss_device") );
```

This is the device instance constructor. It may be called from `sc_main` or the constructor of the top-level module in the design hierarchy. It requires the type of the device to be specified. As a second argument, a user-defined identifier may be passed and is then used for debugging. An example of this statement would be:

```
osss_device my_dev(virtex4, "FPGA_one");
```

This declares a device instance `my_dev` of type `virtex4` and a name `FPGA_one`.

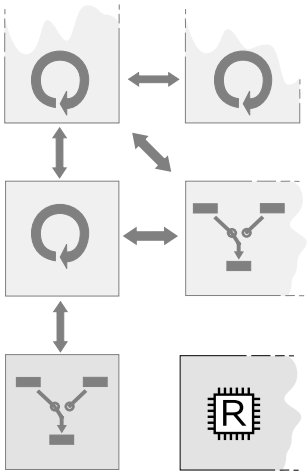

Figure 5.5: Model with reconfiguration control

Binding of Reconfigurable Objects and Devices

So far, it is not decided yet, which reconfigurable object is to be placed onto which device instance. This is done in the next step, as shown in Figure 5.6.

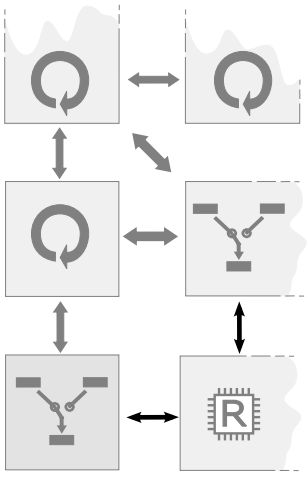

Figure 5.6: Model with control interfaces

```
1 // Previously declared:
2 // - Reconfigurable object "my_ro"
3 // - Device instance "my_dev"
4
5 my_dev.bind( my_ro );
6
7 // or, alternatively:
8 my_dev( my_ro );
```

Again, there are two syntactical different variants that allow the designer to select between a more verbose and a more SystemC-conforming form. With these statements, the number of reconfigurable objects on a device instance can be determined at the end

of SystemC's elaboration phase. This permits instantiating the correct schedulers (with accurate table sizes).

Inheritance Annotation

A final mandatory step is necessary due to a shortcoming of C++. C/C++ does not provide a reflection mechanism to detect the direct parent classes (relation $\lhd^1$) of a given object. This property is essential for the implementation of the OSSS+R library. To fill this gap, the user is required to place a `OSSS_BASE_CLASS` macro inside each of the classes' constructors that are to be used inside polymorphic types. Example for class `B`, `B` $\lhd^1$ `A`:

```
1  class B : public A
2  {
3    ...
4    B()
5    {
6      ...
7      OSSS_BASE_CLASS( A );
8      ...
9    }
10   ...
11 };
```

Second, classes to be used as interfaces for polymorphic types may not be abstract. That means they may not contain pure virtual methods. This makes it impossible for the compiler to check the absence of calls to methods, which are meant to be forbidden. The designer may place a synthesis-compatible marker inside these classes to check at simulation time:

```
1  void MyClass::myMethod()
2  {
3    OSSS_PURE_VIRTUAL_VOID()
4  }
```

Similarly, `OSSS_PURE_VIRTUAL_RETURN()` is available for use in non-void methods. If required, a static check at synthesis time could be added easily. Both macros were introduced due to technical reasons and not due to requirements by the methodology.

5.2 Optional Structural Modeling Elements

The library elements introduced so far are sufficient to describe a basic reconfigurable system. However, there are several ways to optimize a model to increase its performance. The principles are described in Section 4.4 and the relevant modeling elements are introduced next.

5.2.1 Multiple Reconfigurable Areas

In Section 4.4.5 the concept of *slots* was described. The reconfigurable area is logically separated into multiple parts, called slots. Each slot is able to implement the same set of logic.

By default, a reconfigurable object has one slot only. This may be changed by calling this member method on a reconfigurable object:

```
1  void setNumSlots(unsigned int num_slots);
```

If such a call is to be executed, this must be during the elaboration phase and may happen only once. Having multiple reconfigurable areas allows the reconfigurable object to cache contexts, minimizing reconfigurations. But it may also be used to support concurrent accesses to the reconfigurable object. This may be specified by the following method:

```
1 void setNumParallelAccessesAllowed(
2                         unsigned int num_parallel_accesses);
```

By default, there is no access parallelism. The degree of parallelism may be increased to a value up to the number of slots. Regardless of this setting, the modeling library only grants access permissions strictly sequential. New access permissions are granted until all slots are in use or the maximum count for parallel accesses is reached. Nonetheless, accesses may overlap since they may last several cycles and several accesses may be chained without returning permission in-between (with use of lazy permission mode, see Section 4.4). Returning access permission is possible concurrently as well.

5.2.2 Scheduling Algorithms

As explained in Section 4.3.3, a scheduling mechanism is used inside the access control component in case of multiple user-defined processes using the reconfigurable object. By default, this is a round-robin implementation. This simple algorithm guarantees fairness and starvation freedom, although it is not appropriate in all models. Therefore, OSSS+R allows user-defined scheduling algorithms to be used.

New scheduling algorithms need to be derived from predefined base-classes provided by the library. For access control scheduling algorithms, the required base class is `osss_recon_scheduler` and for reconfiguration control scheduling algorithms is `osss_device_scheduler`.

Access Control Scheduler

The `osss_recon_scheduler` class provides a pure virtual method `schedule()` (without arguments) that needs to be overwritten. This method is being called whenever a scheduling decision is to be made and at least one request is valid. This includes cases where exactly one request is valid only. Otherwise schedulers would not be able to maintain a complete history of requests and grants. It is required that the scheduler picks a legal schedule and returns its index value (scheduling needs to be successful in all cases).

Second, there is a data member `jobs` that provides information about the current situation whenever `schedule()` is called. The type of `jobs` is `osss_job_array`, which is defined as follows:

```
1 class osss_job_array
2 {
3 public:
4   const osss_recon_job_descriptor&
5   operator[](unsigned int index) const;
6 };
```

The only operator `[]` allows accessing `jobs` by translating an index value (starting with 0) into a read-only structure that describes the state of a user-defined process. Each user-defined process may have a pending request for the access controller to service. The information found in the job table is described in detail in Section 4.3.3. The user-defined processes are being assigned indices in this table, that are not in any particular order but remain fixed during runtime.

This table is not the only source of information for the scheduling algorithm. As an example, the default round-robin descriptor stores its last scheduling decision in a data

member. Similarly, an user-defined scheduling algorithm may store history information or advance planning in its data members.

In addition, there are a number of auxiliary member functions that provide essential information about the model and the reconfigurable object in particular:

```
unsigned int getNumberOfContexts() const;
```

This method yields the number of contexts that are bound to the reconfigurable object. The value does not change during runtime. It includes the temporary context, if it may be created. Actually, creating the temporary context does not increase this value.

```
unsigned int getNumberOfClasses() const;
```

This method returns the number of user-defined classes that may possibly be assigned to this reconfigurable object.

```
unsigned int getNumberOfUserProcesses() const;
```

The method returns the number of user-defined processes declared to use this reconfigurable object. It also determines the size of the job table.

```
bool isConfigurationControllerBusy() const;
```

This method allows checking the state of the device's configuration control. It may be used by `schedule()` to prefer jobs, which do not require reconfigurations while the resource is blocked.

Reconfiguration Control Scheduler

The `osss_device_scheduler` is very similar to the access control scheduler base class. It also provides a pure-virtual `schedule()` method as a hook for user-defined code. There is also a job descriptor table `jobs` with an operator `[]`. The job table entries differ, their contents are shown in Section 4.3.5.

There is only one auxiliary method required:

```
unsigned int getNumberOfReconObjects() const;
```

This may be used to determine the amount of access controllers connected to this device configuration controller.

Applying User-Defined Scheduling Algorithms

Having implemented custom scheduling algorithms, they may be applied by invoking the `setScheduler()` method on reconfigurable objects or device instances.

```
// Selecting scheduling class "A" for
// reconfigurable object "my_ro"
my_ro.setScheduler< A >();

// Selecting scheduling class "B" for
// device instance "my_dev"
my_dev.setScheduler< B >();
```

The selection of a scheduling algorithm for a reconfigurable object needs to be placed in the constructor of the module containing the reconfigurable object. This way it is executed during elaboration. The selection of a device instance scheduler is to be placed in the constructor of the top-level module or inside `sc_main()`.

5.2.3 User-Defined Timing Datatype

Access schedulers and reconfiguration schedulers may base their decisions on the duration of configuration processes, state preservation processes, and state restauration processes. The OSSS+R infrastructure provides these values in terms of clock cycles to the scheduling algorithms. For some systems such a cycle-accurate information is desireable. For others, a rough indication would suffice and a precise representation would waste hardware resources. Therefore, the datatype used to represent these values is customizable. By default, a predefined class named `osss_precise_cycle_count` is used, which refers to a 32 bit integer. Using `setCycleCounter()` the designer may specify other datatypes. These are not just restricted to shorter integers but may be user defined classes (e.g. only representing powers of two).

```
template< class CycleCountClass >
void setCycleCounter();

// Examples:
my_recon_obj.setCycleCounter< MyCustomDatatype >();
my_device_instance.setCycleCounter< MyOtherDatatype >();
```

All cycle-counter datatypes must provide a specific interface, which is defined in the base-class `osss_cycle_count`. To allow mixing arbitrary scheduling algorithms with such custom datatypes, having such a fixed method interface is mandatory. The interface provides methods to convert integer values to the custom datatype and back, as well as, some primitive operations directly on the custom datatype itself.

The methods of this interface that need to be implemented are:

```
// Converts integer type to custom representation
explicit
osss_precise_cycle_count(sc_dt::uint64 cycles);

// Converts custom representation to integer type
virtual
sc_dt::uint64
getCycleCount() const = 0;
```

The methods that may be overwritten are:

```
// Adds two custom types
virtual
osss_cycle_count &
operator+(const osss_polymorphic<osss_cycle_count> &);

// Comparison operators
bool operator<(const osss_cycle_count &) const;
bool operator>(const osss_cycle_count &) const;
bool operator==(const osss_cycle_count &) const;
bool operator!=(const osss_cycle_count &) const;

// Increases the number of cycles by the given value
// There is no check against wrap-arounds.
virtual
void
increase(sc_dt::uint64 by_cycles);

// Decreases the number of cycles by the given value.
// There is no check against wrap-arounds.
virtual
```

```
21 void
22 decrease(sc_dt::uint64 by_cycles);
```

Note that the modification may not exactly increase or decrease by the given amount. If less precise internal representations for a cycle count value are chosen, they may be unable to reflect the change. For example, storing the logarithm of a number (rounded to the next integer) instead of the plain value saves space. Decreasing 2^4 by one and rounding the result yields 2^4 again. One should keep this in mind when doing calculations on cycle count values.

Without overwriting these, the custom representation if converted to an integer type before the operation and back into the custom representation after it. This can be avoided by overwriting these methods, potentially saving some further logic gates, increasing simulation performance and avoiding rounding issues.

All values representing a process duration (e.g. a reconfiguration or a state restauration process) presented to the scheduler instances inside the job tables are encoded in the selected cycle-count datatype. The access scheduler may instantiate this datatype without knowing the selected implementation by means of a member method of its base class. Both `osss_recon_scheduler` and `osss_device_scheduler` provide the following method:

```
1 osss_polymorphic< osss_cycle_count >
2 getCycleValue( sc_dt::uint64 value ) const;
```

The method returns a polymorphic type, since the cycle-count time is not known at compile-time of the simulation model. This means that a straightforward implementation would not save anything. The reason is that the synthesis tool must implement the complete inheritance tree from the `osss_cycle_count` base class on, including the predefined `osss_precise_cycle_count` class. However, since the dynamic type of the cycle count class is known at synthesis time, the datatype may be replaced by a non-polymorphic one: the one class that was selected by the designer.

5.2.4 User-Defined Placement Algorithms

For each reconfigurable object that has more than one slot (that means, a reconfigurable area logically split up in separate parts) a further algorithm is required. Whenever, a context is to be enabled, a slot needs to be selected to place the logic. The default placement algorithm is described in Section 4.4.5. Like the default scheduling algorithm, a different algorithm may yield better results for a given application. To implement such a custom algorithm another base class is provided: `osss_placer`. The placement method is defined as follows:

```
1 virtual
2 unsigned int
3 place( const osss_context_descriptor& to_be_placed) = 0;
```

It is invoked for each context that requires a placement decision. The timing implications of that decision are then calculated and presented to the scheduler to aid the decision. If the scheduler decides not to enable the context, the request is not being serviced. Then, another placement decision needs to be made prior to the next scheduling decision. In other words, a placement decision is not kept infinitely, but re-calculated each time a context may potentially become enabled. The reason is that enabling one context might include reconfigurations and therefore, significantly change the basis for a placement decision, especially if the re-use of FPGA configurations is a goal.

The placement algorithm may rely on a number of auxiliary methods provided by `osss_placer`:

```
1 unsigned int getNumberOfContexts() const;
2 unsigned int getNumberOfClasses() const;
```

```
3 unsigned int getNumberOfSlots() const;
```

These methods are intended to allow access to some important constant values. These are the number of contexts bound to this reconfigurable object (including the temporary one, if it exists), the total number of classes that might be configured to a slot, and the number of slots.

To give the placement algorithm a foundation for its decision, it has access to a description table for all slots by means of a data member `jobs`. Like its siblings in the scheduling base classes, an operator `[]` gives read-only access to the fields of a table. The contents provided for each slot are explained in Section 4.4.5.

The context for which a placement decision is required is described by the argument passed to the `place()` method. It provides information about whether the context is currently enabled or not. If it is enabled, the location (slot index) is available. Furthermore, it indicates whether the context is locked and what is its current dynamic type.

A scheduling algorithm may rely on at least one job being valid. The placement algorithm, however, must cope with the situation that it has to decide about a context for which no appropriate placement decision can be made. The algorithm is then free to return a special illegal value (the number of slots). This value will cause all requests for the context to be marked as currently being invalid.

The placement algorithm has access to the reconfiguration controller's busy state. By taking this value into account, a placement algorithm may avoid reconfigurations.

```
1 const bool
2 isConfigurationControllerBusy() const;
```

Consider a set of contexts having the same class to be placed, only one slot matching that class, and a busy reconfiguration controller. Placing all contexts to this only matching slot may cause frequent context saves and context restores. On the other hand, it avoids reconfiguration. Observing the reconfiguration controller to switch from busy to ready state, could make it worth spending some reconfigurations to minimize context saves and restores. One cannot decide which strategy is best without taking the application into account.

A custom placement algorithm may be installed as shown in the following example:

```
1 my_recon_obj.setPlacer< MyPlacerClass >();
```

5.3 Object Manipulation Modeling Elements

The first sections of this chapter explained how to specify structural properties of model in OSSS+R. This section focuses on the dynamic part, that is, the operations performed at runtime. The operations are introduced with their syntax and restriction. Their timing is specified separately, since one may implement a different underlying infrastructure to execute the same operations. Such a different infrastructure is likely to show a different timely behavior.

5.3.1 Operations on Reconfigurable Objects

First, the operations on the anonymous access object are explained (see Section 4.3). OSSS+R does not introduce another identifier for this purpose, but re-uses the identifier of the reconfigurable object itself. Anonymous accesses are permitted for one-slot reconfigurable objects only (see Section 4.4.5).

Since the following methods represent operations of multiple clock cycles length, they are only to be used inside the bodies of clocked SystemC threads (`SC_CTHREAD`s).

Assignment Operators

```
1  // Assignment operator for identical class
2  void operator=( osss_recon< InterfaceT > &);
3
4  // Assignment operator for recon objects
5  template< class DerivedT >
6  void operator=( osss_recon< DerivedT > &);
7
8  // Assignment operator for permanent contexts
9  template < class DerivedT >
10 void operator=( osss_context< DerivedT > &);
11
12 // Assignment operator for polymorphic objects
13 template < class DerivedT >
14 void operator=( const osss_polymorphic< DerivedT > &);
15
16 // Assignment operator for plain user classes
17 void operator=( const InterfaceT &);
```

These operators allow anonymous assignments to the reconfigurable object. The right-hand side may be another reconfigurable object, a permanent context or a polymorphic object. In all three cases, the interface class must match those of the reconfigurable object or needs to be derived from it. Finally, assigning any class, which is identical to the reconfigurable object's interface class or derived from it is permitted.

Anonymously assigning to a reconfigurable object causes a reconfiguration if the new dynamic type differs from the previous dynamic type present in the slot. Also, it may cause saving a context's state, if the destination slot is occupied by a permanent context. Since, anonymous assigns always target the temporary context, this never causes a state restauration.

An assignment is essentially a method call to the right-hand dynamic classes' copy operator. It is combined with requesting a change in the dynamic class type. The steps performed by the user process are:

1. If in lazy permission mode for that reconfigurable object and currently holding any permission, return it.
2. Request access permission and dynamic class adaption, simultaneously from access controller for the anonymous access object.
3. Perform copy-assignment method call.
4. If not in lazy mode, return the permission.

In case of assigning a value object (plain or polymorphic) to the reconfigurable object, this is sufficient. Assigning another reconfigurable object or a named context requires reading its value prior to the assignment. Reading this value is a method call, which is performed first (see `operator->` below). Since both reconfigurable objects and permanent contexts are polymorphic in nature, the conversion method to a polymorphic object is used to read their values (see `operator osss_polymorphic< InterfaceT >()` below).

Timing. In case of multiple user-defined processes using the reconfigurable object the timing is as follows:

Cycles	Action
0 or 1	Return permission (if required, see above).
1	Indicating permission request and dynamic class adaption to access controller.
$n \geq 2$	Awaiting acknowledge indicating completion of process
1	Requesting execution of the assignment method call from slot.
$1 + m$	Await method completion, this takes at least one cycle. It may be extended by m cycles, if the copy-assignment operator contains `wait()` statements (which is highly discouraged by OSSS methodology).
0 or 1	Return permission (if not lazy).

In case of a single user-defined process using the reconfigurable object, this process is simplified, since permission handling is not necessary:

Cycles	Action
1	Requesting class adaption from to access controller.
$n \geq 2$	Awaiting acknowledge indicating completion of the process.
1	Requesting execution of the assignment method call from slot.
$1 + m$	Await method completion (see above).

Optimized Assignment (In-Place Construction)

```
1 template< class RHSType >
2 void construct();
```

This is a special function, which may be used if the right-hand side of the assignment is intended to be a default-constructed object. It represents a reconfiguration of the dynamic class, if required. Once the correct dynamic class is in place, the default constructor is executed. It is functionally equivalent to an assignment of `RHSType()` to the reconfigurable object but may be faster.

Timing. `construct()` allows an implementation, which is optimized compared to a regular assignment. The knowledge that the desired dynamic state after the call matches those of an object after default construction may be used. The OSSS+R modeling library assumes that any content of a slot can be quickly reset to its default-constructed state by the access controller. This is done each time a dynamic state switch is requested, regardless of a class change being performed or not. A construct call may then omit the copy-assignment method call, since the anonymous access object is already in default state.

Method Invocation

```
1 T& operator->();
```

The arrow operator can be used to manipulate the anonymous access object. `T` is the interface class type for the reconfigurable object. For example, `rval = aao->m( myarg );` represents an access to the reconfigurable object `aao`, calling the method `m` with argument `myarg` and a return value `rval`. Method invocation does not only support return values to hand back data, but reference parameters, as well. They are treated with a copy-in-copy-out semantics.

Timing. In case of multiple user-defined processes using the reconfigurable object:

Cycles	Action
0 or 1	If not in lazy mode or not holding a permission: Request access permission.
0 or $n \geq 2$	If requested permission, await acknowledgement from access controller.
1	Requesting the execution of method call from slot.
$1 + m$	Await the method completion, this takes at least one cycle. It may be extended by m cycles, if the method contains `wait()` statements.
0 or 1	Return permission (if not lazy).

This is drastically simplified in case of a single using process. There is no need for interaction with the access controller.

Cycles	Action
1	Requesting the execution of method call from slot.
$1 + m$	Await the method completion. This takes at least one cycle. It may be extended by m cycles, if the method contains `wait()` statements.

State Reading

```
1 operator osss_polymorphic< T >();
2
3 template< class TargetType >
4 TargetType cast();
```

These two methods allow reading the dynamic state of the anonymous access object. The first one is an implicit conversion to a polymorphic object having the same interface class type than the reconfigurable object. The second one, `cast()`, is an explicit type-cast function to a non-polymorphic type. The second function is just for convenience, it uses the first function internally without adding extra timing overhead.
Timing. The implicit conversion function has the timing of a method call. It executes the copy-assignment operation without altering the anonymous access object.

If copy-assignment operators are implemented to execute `wait()` statements, this method cannot provide exact timing. In fact, the access permission is returned (if required) before the assignment operator is executed. This is due to technical reasons in the C++ library.

On the other hand, placing `wait()` statements inside copy-assignment operators is dangerous. SystemC is based on C/C++ which makes intense use of creating and destroying temporary objects for constness reasons or casting. This results into a process behavior, which is very difficult to predict for a human. A second drawback is that copying such classes would only be possible inside clocked threads. Other types of SystemC processes do not allow executing `wait()` calls. For these two reasons, OSSS (not just OSSS+R) discourages the designer to use `wait()` statements inside copy constructors at all.

5.3.2 Operations on Persistent Contexts

Regarding syntax, operations on the anonymous access object and persistent contexts are specified the same way. However, persistent contexts (see Section 4.3.2) allow a more abstract modeling and are mandatory for multi-slot reconfigurable objects.

Assignment Operators

The following operators allow assigning other permanent contexts, reconfigurable objects (via anonymous access object read), polymorphic objects, and plain user defined objects to a permanent context:

```
1  // Assignment operator for other contexts
2  template < class DerivedInterfaceT >
3  osss_context< T >&
4  operator=( osss_context< DerivedInterfaceT > &);
5
6  // Assignment operator for recon objects
7  template< class DerivedInterfaceT >
8  osss_context< T >&
9  operator=( osss_recon< DerivedInterfaceT > &);
10
11 // Assignment operator for polymorphic objects
12 osss_context< T >&
13 operator=( const osss_polymorphic< T > &);
14
15 // Assignment operator for plain user classes
16 osss_context< T >&
17 operator=( const T &);
```

The steps to be performed are identical to the assignments to the anonymous access object, except for one particular situation. If the lazy permission mode is used, an access permission is currently held by the user-defined process, and the assignment was obtained anonymously, the permission needs to be returned. The reason is that the user-defined process does not know on which context a permission was granted in case of an anonymous permission request. If a permanent context is addressed afterwards, this is done using a named access instead of an anonymous access. It is unknown to the user-defined process whether the permission, which it currently holds matches the new permanent context.

If the previous permission was obtained using a permanent context's name, the user-defined process is able to tell whether the new named context to be accessed matches the previous one. If so, the old permission may be re-used. Otherwise, it needs to be returned, too.

Timing. In the case of multiple user-defined processes using the reconfigurable object, the timing is identical to the assignment operators of the anonymous access object. The same protocol is even used in case of only one user-defined process. The reason for not using the optimized protocol without permission handling is that permissions are used to control context switches. Returning and re-requesting permissions in case of context switches enabled the access controller to actually perform these necessary switches.

Optimized Assignment (In-Place Construction)

```
1  template< class RHSType >
2  void construct();
```

This method allows in-place default construction of permanent contexts. First, if in lazy permission mode, non-matching permissions need to be returned. This is the case, if the previously obtained permission refers to a different permanent context or

was requested anonymously. Then, the new permission needs to be requested, if it is not re-used from the last access.

Timing. The timing for the `construct()` method is:

Cycles	Action
0 or 1	Return permission, if required.
1	Request in-place construction.
	Additionally request access permission, if required.
$n \geq 2$	Await acknowledge from access controller.
0 or 1	Return permission (if not lazy).

Method Invocation

```
1 T& operator->();
```

This operator allows the invocation of methods on a permanent context having the interface class type `T`. Unlike the operator on the anonymous access object, this operator first checks whether an old permission needs to be returned in lazy permission mode. If necessary, the permission is automatically returned.

Timing. In case of multiple user-defined processes using the reconfigurable object:

Cycles	Action
0 or 1	If in lazy mode and holding a permission that is not known to match this permanent context: Return old permission.
0 or 1	If not in lazy mode or not holding a permission: Request access permission.
0 or $n \geq 2$	If requested permission, await acknowledgement from access controller.
1	Requesting execution of method call from slot.
$1 + m$	Await method completion. This takes at least one cycle. It may be extended by m cycles, if the method contains `wait()` statements.
0 or 1	Return permission (if not lazy).

Explicit Enable

```
1 virtual void enable();
```

This allows explicitly enabling a context without performing an operation on it. It refers to requesting access permission and returning it immediately afterwards. It is useful in combination with external locks to implement context prefetching.

State Reading

```
1 operator const osss_polymorphic< T > ();
```

This method provides a type conversion that enables assignments from context objects to normal objects (but only iff assignment-compatible).

5.3.3 Locking Mechanisms

In Section 4.4.3, three different scenarios and appropriate locking concepts are presented. The concepts were motivated using a cryptography example with two processes u_0 and u_1 and each of them exclusively using a persistent context (named c_0 and c_1,

respectively). Both contexts are bound to the same reconfigurable object r. The very same example is used here to demonstrate the library elements to implement the various locking mechanisms.

Scenario A (External Locks). In the first scenario, it is desired to prevent a persistent context from being disabled for a certain sequence of steps. This concept is named *external lock*. The following code fraction shows a fraction of the user-defined process u_0 (named `process0`) implementing such a lock on c_0 (named `context0`).

```
1  void process0()
2  {
3    ...
4    OSSS_KEEP_ENABLED( context0 )
5    {
6      // The lock is in effect inside this code block
7      ...
8      context0->setKey( ... );
9      context0->initialize();
10     encrypted_data = context0->encrypt( plain_data );
11     ...
12   }
13   // Here the lock returns to false
14   ...
15 }
```

The lock on `context0` is active, while the control flow of `process0` is inside the `OSSS_KEEP_ENABLED` block. Entering this block does not enable `context0`. This is done when the first manipulation (`setKey()`) is executed, forcing the context to become enabled. Once it is enabled, it will not become disabled again, until the `OSSS_KEEP_ENABLED` block is left. Leaving the block does not necessarily disable the context. It rather allows it to become disabled due to manipulations of other contexts (for example, c_1 by u_1).

Nesting `OSSS_KEEP_ENABLED` for the same context is not permitted.

For `OSSS_KEEP_ENABLED`, safety considerations lead to a block-statement implementation. An alternative way of expressing such a locking mechanism would have been to provide `lock()` and `unlock()` methods to be executed on persistent contexts. However, a flawed control flow bypassing an `unlock()` statement after locking a context may be difficult to detect. In contrast, automatically unlocking a context when leaving the scope inside, which was locked is more restrictive but considerably safer.

Scenario B (Swift and Lazy Modes). The second scenario was comparable to the first, but with one major difference: The locked context should be reserved for exclusive access. In contrast, external locks still permit shared access to a context.

From a technical perspective, exclusive access is very easy to realize in OSSS+R. User-defined processes always compete for access permission before any context manipulation. Inhibiting access grant to other processes is an effective way to implement locking. In default mode (called *swift mode*), processes return access permission immediately after use. By simply retaining the access permission over a sequence of manipulations, an exclusive access is guaranteed. This mode is called *lazy mode*.

The library implementation is very similar to those for external locks:

```
1  void process0()
2  {
3    ...
4    OSSS_KEEP_PERMISSION( context0 )
5    {
```

```
6       ...
7       // This first context manipulation requests the
8       // permission:
9       context0->setKey( ... );
10      context0->initialize();
11      encrypted_data = context0->encrypt( plain_data );
12      ...
13    }
14    // Leaving the block causes retained permissions
15    // to be returned, if any.
16    ...
17  }
```

Entering the `OSSS_KEEP_PERMISSION` block statement does not request any access permission. The permission is obtained due to the first context manipulation inside the block. As long as the very same context is used and the block is not being left, the permission is retained.

There is a major difference between external locks and a user-defined processes' swift mode. While locks always refer to exactly one persistent context, the lazy permission mode is enabled for all contexts bound to a reconfigurable object. The reconfigurable object is determined by the binding of the context. Manipulating a different context of the same reconfigurable object inside the `OSSS_KEEP_PERMISSION` block will return the retained permission, and make the process request a new permission for the other context. This is necessary, since each user-defined process may hold one permission at a time for each reconfigurable object only.

It is permitted to nest `OSSS_KEEP_PERMISSION` blocks for different reconfigurable objects. Nesting these for the same reconfigurable object is disallowed. To ease implementations, it is also permitted to directly pass a reconfigurable object (instead of an context identifier), at the header of the block.

Scenario C (Internal Locks). The third scenario was about a context state indicating whether it was appropriate to lock the context or not. One solution could be to monitor the context's state and use external locking to react on the context's state. This would be overly complicated and likely a maintenance issue making modifications difficult.

An internal locking mechanism provides a cleaner solution. A release guard is specified by means of a Boolean equation, acting as a necessary condition to disable a context. If this guard condition evaluates to false, disabling the context is inhibited.

If a class does not have a release guard declaration, the guard is implicitly treated as being `true`. In OSSS+R, defining a release guard inside a class is done using the `OSSS_NEW_RELEASE_GUARD()` macro. It requires an argument being a Boolean expression, as shown in this example:

```
1  class MyClass
2  {
3    private:
4    int buffer[40];
5    int fill; // number of elements in buffer
6
7    OSSS_NEW_RELEASE_GUARD( fill == 0 );
8
9    ...
10 };
```

This would inhibit disabling any contexts of dynamic type `MyClass`, while the buffer is non-empty. Derived classes may add further conditions that are required in ad-

dition to the inherited conditions. It is done using the `OSSS_AND_RELEASE_GUARD()` macro:

```
class MyDerivedClass : public MyClass
{
  bool connection_established;
  OSSS_AND_RELEASE_GUARD( ! connection_established );
  ...
```

This introduces a further requirement of `connection_established` being `false` in order to disable a class. Finally, a derived class may add conditions, which permit disabling a context in special cases:

```
class MyDerivedClass : public MyClass
{
  bool buffer_may_be_cleared;
  OSSS_OR_RELEASE_GUARD( buffer_may_be_cleared );
  ...
```

The resulting equation would be `(fill == 0) || buffer_may_be_cleared`.

5.3.4 Transient Attributes

Once a class is properly guarded, the state information relevant for saving and restoring a context may be restricted. This is done by introducing transient attributes, see Section 4.4.4. Consider the `MyClass` example again:

```
class MyClass
{
  private:
  int buffer[40];
  int fill; // number of elements in buffer

  OSSS_NEW_RELEASE_GUARD( fill == 0 );
  OSSS_TRANSIENT_ARRAY( buffer );
  ...
};
```

The annotation `OSSS_TRANSIENT_ARRAY( buffer );` makes `buffer` transient. Upon context saving, this data member would be omitted. After restoring the context, the entries in the `buffer` are undefined. But, if a `MyClass` is omitted in cases of `fill` being zero, we do not need to save `fill`, too. This can be declared using `OSSS_TRANSIENT()`:

```
class MyClass
{
  private:
  int buffer[40];
  int fill; // number of elements in buffer

  OSSS_NEW_RELEASE_GUARD( fill == 0 );
  OSSS_TRANSIENT_ARRAY( buffer );
  OSSS_TRANSIENT( int, fill, 0 );
  ...
};
```

This way, `fill` would be transient, too. After restoring a context of type `MyClass`, its `fill` member would be set to zero.

5.4 Possible Extensions

This chapter introduced one consistent set of object properties, restrictions and timing. The modeling elements were designed in parallel to a synthesis concept for each of these. Restrictions by the C++ language have an influence on the modeling as well as synthesis considerations.

However, this is just one way of implementing the concepts presented in the Chapter 4. A few possible extensions are suggested next to give some ideas for further developments.

Protocol Modifications. There could be different timing schemes possible, for example, one could restrict the bitwidth of connections to trade in runtime for area. Different handshaking schemes (or even partly synchronous protocols) could further fine-tune system's runtime behavior. These changes would necessarily not affect the modeling style in general. One could still use method invocations to manipulate reconfigurable objects. These manipulations would probably show a different timing because they would cause a different communication, but they could still be expressed as method invocations.

Another promising extension would be to utilize the recent achievements of OSSS in terms of the communication modeling for OSSS+R [55, 28].

Realtime Support. Besides timing decisions at the clock-cycle level, more coarse changes would be possible, too. One decision was to have runtime schedulers deciding about the use of limited resources at runtime. Runtime scheduling is likely to give good scheduling decisions in the average case. But since runtime scheduling is very difficult to predict statically (by nature), it is not very suitable if hard timing-constraints have to be met. In contrast, a time-division multiplex access scheme (TDMA) for critical resources with fixed time slots would better support realtime-constrained systems.

Support for Field Maintenance. Applying updates, while the system is in the field and in operation yields its specific challenges. First, the update mechanism itself needs to be implemented. Updates need to be applied atomically, requiring a double-buffered configuration memory. Otherwise, reconfigurations need to be inhibited, while the update is in progress. Further challenges arise from the design itself. Only dynamic parts of a design may be updated (by definition). The updates may then break the compatibility between the static and dynamic parts. As soon as an update modifies a datatype used in both static and dynamic design parts, the update needs to be crafted very carefully. A set of rules may help, e.g. allowing datatype modifications, if operations on that type are only executed in the dynamic part. This would allow modifying method implementations. Additionally, the state space of a type may be extended, if new data members are declared transient and are properly guarded by internal locks.

5.5 Chapter Summary

This chapter presented the modeling elements of the OSSS+R simulation library for concepts explained in Chapter 4. First, the essential steps to specify a reconfigurable model were introduced. Each step was illustrated with a block diagram sketching what part of the resulting system was described. Second, optional structural modeling elements were presented, for example custom scheduling or placement algorithms. The third part covered the runtime operations and their optimization. Finally, some potential extensions were sketched to give some directions for future developments of OSSS+R.

Chapter 6

Synthesis of OSSS+R Models

In this chapter, the synthesis of OSSS+R systems is described. The first part explains the synthesis process itself. The input analysis is explained here and combined with an overview of OSSS+R specific transformations. Next, the register transfer level components that replace OSSS+R elements are described. The focus of the second part is on how these replacement components interact and implement a functionality that is equivalent to the former OSSS+R models.

6.1 Synthesis Process

6.1.1 SystemC and OSSS Synthesis

SystemC was not standardized as a language, but as a software library. When it comes to synthesizing SystemC, it becomes obvious that this is more than a subtle difference. When designing a hardware description language, it is useful to make it suitable for analysis by tools, unambiguous and well-defined. To some extent this is a conflict to its C++ roots, for example when it comes to the use of pointers, which are difficult to analzse for a tool in the general case. Other oddities were introduced that make sense from the perspective of a software library only. There are cases where the bitwidth of a shift operation's result depends on the distance on which the first operand is being shifted. Other operations yield the results having the size of the first argument, if the second operand of a multiplication is one and different sizes in other cases. From the perspective of a software library, it makes sense to detect corner-cases (e.g. multiplication by one) and use simplified computations (e.g. directly returning the operand being multiplied by one). Since, SystemC is designed for high-speed simulation of abstract models, this can be justified, as long as no unexpected behavior occurs.

For a hardware synthesis tool, these optimizations are rather obstacles than advantages. To exactly represent the behavior of a SystemC model in another language (e.g. VHDL) the operators of SystemC datatypes need to be modelled preserving their rough edges. Otherwise, mismatches of simulation and synthesis could occur, which are very difficult to detect.

fossy is a tool designed to accomplish this task. Its purpose is to transform an OSSS model into a pure register-transfer description, which can be expressed using VHDL. OSSS itself is a synthesizable subset of SystemC, which is extended by additional elements to ease the transformation of C++ models into hardware. Furthermore, *fossy* accepts user-defined types (e.g. classes, structs, or unions) and flattens these to VHDL records and bitvector types. Operations on these types are converted, too. The transformations inside *fossy* can be separated into different phases.

C/C++ Front-End Phase. First, the SystemC source code is read using a commercial C++ front-end[2]. Besides lexical analysis and parsing, it performs some more sophisticated preparations, too. For example, it assigns unique identification numbers to identifier symbols. This makes it easier to distinguish identifier symbols which are given the same name but located in different scopes. Another example is the folding of some constant values or the insertion of explicit casts, wherever C++ implies one.

SystemC library elements are treated in a special way. Instead of analyzing the SystemC software library along with the input model, a specially crafted set of include files provides all synthesizable SystemC symbols. For these symbols, special treatment is implemented in later phases of *fossy* . After these processing steps, C++ templates are instantiated explicitly. SystemC integer types and bitvector types are implemented as templates and also receive their bitwidths in this step.

Once the model was analyzed completely, it is exported in a XML-based file format. While this first phase of *fossy* is implemented as C code, all subsequent phases are implemented in the Haskell[3] language. To bridge this language gap the XML file containing the syntax tree is imported into the Haskell part again.

Elaboration. The purpose of the elaboration is to identify SystemC language elements. From a syntactical perspective, the syntax tree contains a C++ model, which makes use of SystemC elements. These model elements are now identified and made explicit, which is called *elaboration*. For example, bitvector types are found to be instances of a `sc_bv` template class type and are now re-written as an internal type explicitly representing a sequence of bits (instead of being an instance of a class).

Then, the top-level instance of the design is identified. Starting with this instance, a design hierarchy is built by recursively identifying sub-instances. For each module, special SystemC members like ports and signals are transformed from being data members of their template types (e.g. `sc_signal`) into module elements of their respective interpretation (e.g. being a signal). The same elaboration is used for OSSS elements, e.g. shared objects and polymorphic objects.

Synthesis of OSSS Elements. This phase is to resolve OSSS modeling elements. A replacement circuitry is calculated and instantiated, where the OSSS element was used. This also includes the generation of communication links, transformation of operations and special datatype containers.

The replacements need not necessarily to be representable by single VHDL primitives. They may also be more complex structures. Instead, this phase relies on successive phases to further process even these replacement circuits. For example, polymorphic types are resolved into a structure having a tag and a payload member. The tag identifies the dynamic type information and the payload member provides the data to be interpreted according to the tag value. The data member itself is of a C `union` type having all the possible variants as its members. Operations on polymorphic objects are replaced by a switch over the tag value. Depending on each case, a different runtime type is used, which can be determined statically. For each case, a different manipulation is inserted with a resolved dynamic type. Unions, however, cannot be expressed in VHDL. Still, they are generated in this phase and left as a task for further processing steps.

SystemC Synthesis. SystemC synthesis has datatype transformation as its primary purpose. The class inheritance tree is being flattened, enriching child classes with elements from their parents and removing the inheritance dependency. Virtual method calls are being resolved, since all use of dynamic types is now unfolded into the control flow and made explicit.

Then, methods are transformed into free C functions, accepting the former object as their first parameter. The object itself is now an instance of a C structure and no

longer a class having member methods.

Operations on SystemC datatypes are made explicit, including implementations of all rough edges. Some corner cases cannot be supported, since there is no hardware representation making sense. Examples are SystemC integers, which can be constructed in different ways, representing the same value but still behaving differently on some operations.

After this phase, the model is on a register-transfer level of abstraction. Only signal-level communication remains but no method-based communication.

Postprocessing and Code Generation Although register transfer level is reached, some further processing is necessary depending on the output format. To give an example for these transformations, identifiers are adjusted to VHDL naming conversions. Once this is finished, *fossy* exports the transformed model.

6.1.2 Design Decisions

Before the synthesis process itself is described, some fundamental design decisions are discussed.

Context State Preservation. For permanent contexts, it is necessary to make them appear to be virtually permanent, as the name implies. This can be done in several ways.

First, using a chip that has a set of dynamic states and is able to switch among them during runtime. Switching both logic and dynamic state was proposed for FPGAs. These devices are called multi-context FPGAs. However, none of the commercial successful FPGA families support multiple contexts. Relying on this feature would make OSSS+R unusable for these devices, which is not an option.

The second, natural solution is to save and restore the dynamic state of reconfigurable areas by implementing configuration read-back. While others [61] use this solution, it is also a restriction. The dynamic state needs to be extracted [137] from the configuration stream and merged again into the configuration stream during restoration. This requires extra hardware effort and runtime. A configuration readback is likely to be comparably slow as a reconfiguration and restoring a dynamic state would require another reconfiguration. So, this would only be a solution for extremely infrequent object switches. A final drawback is that not all systems can be assumed to allow the readback of FPGA data. There are works on encrypting configuration streams and decrypting them on-chip during the reconfiguration process itself [131, 58]. Some FPGA devices already have built-in support for decryption, e.g. recent Virtex FPGA families[142, 141, 146]. Such systems are unlikely to allow configuration readback, since it would break protection of intellectual properties.

The third (and chosen) way to implement context state preservation is to embed this process into the application logic. The implementations inside the reconfigurable areas are equipped with an interface that not only allows to initiate method executions, but also dynamic state reading and writing. No reconfiguration is required when reading and writing a state. A drawback is that it alse causes hardware overhead. This is rooted in the requirement to make all flip-flops that store class attributes accessible from the save-and-restore control component. Additional wires are required, which in turn are a critical and limited resource in todays FPGAs. Flip-flops that do not store class attribute data (e.g. local variables or control flow information) are not affected.

Logic Tracking. Another decision affects the tracking of configurations. It would be an appealing idea to introduce a configuration identification port for each slot. The configuration would tie this port to a specific value, uniquely identifying all possible

configurations. This would be a small circuit inside the reconfigurable area, since the port's pins would be tied to zero or one.

There is just one problem: Without having an integrity check at hand, there is no way to tell whether the identification really indicates an operational configuration. Any reset during reconfiguration might leave the signals tied to a special value. And since a reset deletes all dynamic state in the access controllers (which are responsible for keeping track of configurations), those would assume valid configurations to be present. Therefore, it is at least required to maintain a flag which marks all slots as dirty after reset and clean after their first configuration. For prototyping OSSS+R, access controllers are generated to manually track configurations. Honoring a dirty-state flag, this optimization could be applied.

6.1.3 OSSS+R Synthesis

The *fossy* tool was enhanced to support synthesis of a limited set of OSSS+R elements. These are namely:

- Reconfigurable objects (`osss_recon< T >`) with one slot and an arbitrary number of user-defined processes as clients.
- Device instances (`osss_device`) and types (`osss_device_type`).
- Timing declarations (`OSSS_DECLARE_TIME` macro).
- Method invocations on the anonymous access object.
- Anonymous assignments of non-polymorphic objects.
- Default schedulers for reconfigurable objects and device instances.
- Default placement algorithm.
- Default cycle count class use.

For other elements some preparatory work was done, e.g. for permanent contexts or customization of scheduler and placement classes. The components currently generated by the synthesis tool are shown in Figure 6.1. It is almost the same as Figure 4.12.

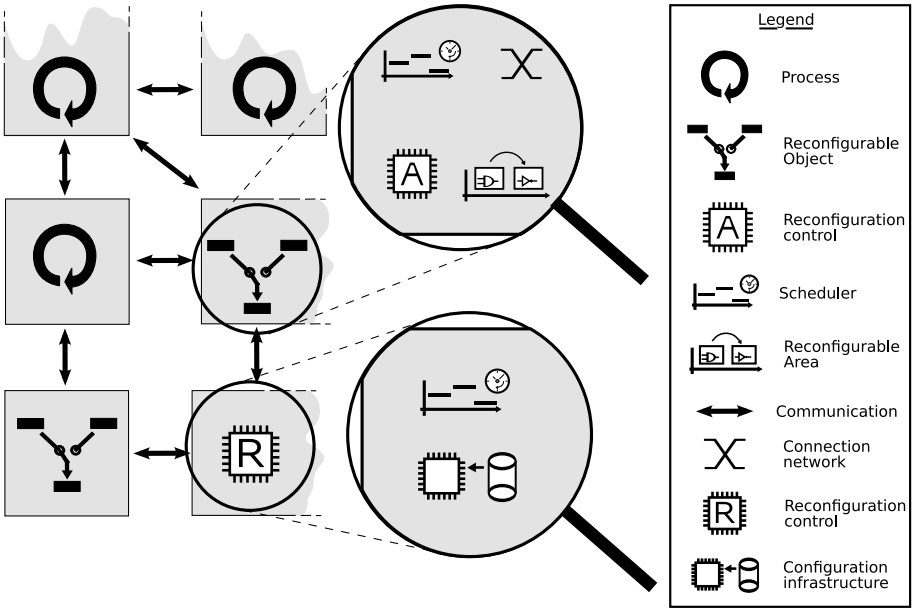

Figure 6.1: Components currently generated by synthesis tool

Only the symbol for the context attribute storage is missing, since it is omly used with persistent contexts. The relation of accepted OSSS+R library elements and the resulting hardware structure was shown in Figures 5.1 to 5.6 (as far as a block diagram was

suitable).

The selected set of supported elements allows some analysis on how much automation can be used in the design process. Furthermore, it allows some estimates of resource costs introduced by the OSSS+R style of modeling applications.

The synthesis of OSSS+R affects three phases of *fossy* 's operation. In the front-end phase, additional library elements are read to support OSSS+R language constructs. Currently, this requires passing `-D OSSS_INCLUDE_RECON` to *fossy* . Inside the elaboration phase, reconfigurable objects, device types, device instances, binding of reconfigurable objects and user processes are elaborated. The transformation of the elaborated OSSS+R elements is done inside the phase for synthesis of OSSS elements.

Figure 6.2 depicts the synthesis phases inside *fossy* with focus on the recon object synthesis. This part is be described next in more detail.

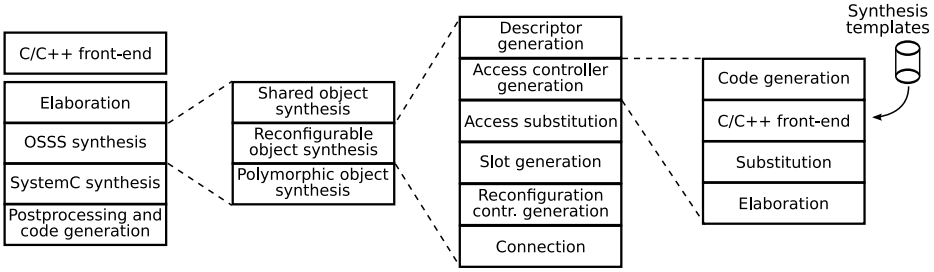

Figure 6.2: Synthesis sequence

Descriptor Generation. There are multiple components in a OSSS+R system, which must be modified in a way that keeps these compatible to each other. To give an example, an user process requesting a context must use the same numbering of contexts as the access controller serving this request. In order to centralize this task, a set of descriptor tables is built. They contain unique enumerations of classes, methods, contexts, etc. For protocols, the port sizes and types are calculated and stored.

Access Controller Generation. This step calculates a circuit to implement the access control of a reconfigurable object. If an access scheduler is required, it will be embedded into the access controller. The scheduling algorithm may be user-defined in principle, therefore it may even be a template instance. History-aware schedulers are likely to maintain data members being arrays of sizes dependent on the number of user processes or template arguments. Scheduler templates, however, are resolved in *fossy* 's front-end phase. During access controller generation, they would still be needed as being templates. To solve this problem, the replacement circuit is first being generated as a SystemC model and exported into the file system. Figure 6.2 shows this as a code generation step. The access control logic that instantiates and uses the scheduler is then imported from the synthesis library. Additionally, the scheduler source code is included. The resulting SystemC code is then processed by the front-end. Here, templates receive the correct arguments and may be safely resolved. This includes the number of clients, interface datatypes, enumerations of methods and other information. The resulting source code is then elaborated with the access controller as its top-level instance. The new instance is then merged into the main design as a replacement for the reconfigurable object instance.

Access Substitution. Next, the user-defined processes are being modified. Method invocations and assignments are searched and replaced by a signal-level protocol. Both

interaction with the access controller and slots is inserted. It is transparent to the user-defined code, whether they are connected directly to the slots or via a crossbar. The descriptors provide the necessary information, which signals are to be inserted or omitted. The reset code inside the user-defined processes is finally extended by reset code for the control signals towards access controller and slots.

Slot Synthesis. The slot synthesis generates the modules which are to be exchanged during runtime. They contain two processes, which listen to commands via the interface to the user-defined process. It is transparent for the slots, if they are connected to the user-defined processes directly or via a crossbar. The processes inside the slots manipulate a data member of the dynamic type, which the slot represents. Slots receive the method identification to be called from the outside, as well as the necessary arguments for the call. After executing the requested method on the internal member, they return the copy-out values to the caller.

Device Synthesis. All access controllers rely on a reconfiguration controller to perform reconfigurations, one per device. This reconfiguration controller consists of a **p**latform **i**ndependent part (PIRC) and a **p**latform **d**ependent part (PDRC). The PDRC is included from a board support package after synthesis and the PIRC is generated in this phase. Similar to the access controller, the component may require a scheduler. This raises the same issues as described with the access controller and is solved in the same way by generating code and reading it back into the *fossy* tool.

Connection All steps of the reconfigurable object synthesis after the descriptor generation and before this step may be executed in an arbitrary order. They are designed to be performed isolated and solely rely on the descriptors. To allow connection of the generated components, each component was instantiated with its respective ports and signals. Additionally, for each port or signal an entry in a global repository is generated. Each entry describes an endpoint of a connection and some look-up criteria to seek its counterparts. In the connection step, groups of endpoints are picked and the required signals and ports to connect the endpoints are generated in the hierarchy. This does not only support point-to-point signals but broadcasts, as well.

6.2 Synthesis Results

The former part of this chapter described the steps being performed by the synthesis tool to generate a register transfer level description. This was restricted to those parts currently implemented in the tool and used for evaluation.

Now, the generated components are described in more detail. The following description presents a set of building-blocks that can be used to implement a complete OSSS+R description on register transfer level. It is not restricted to those components that can be generated by the current version of the synthesis tool.

The components are introduced using a block diagram describing their communication. Each component is shown with their complete set of connections to its communication partners. The partner's connections to third components are omitted. Signals are grouped according to their partner. If a component may have more than one partner of the same kind, the interface is marked with an asterisk (*). Signals, which are mandatory are shown as a solid line, whereas optional signals are shown as dashed lines. The accompanying text describes in which cases these signals are omitted.

Protocols are illustrated using timing diagrams and deterministic finite automata (DFA).

6.2.1 Access Controller

The access controller serves requests from user-defined processes. Processes may request access permission, object switches or changes in a context's dynamic type. The access controller keeps track of reconfigurations and dynamic types for all contexts.

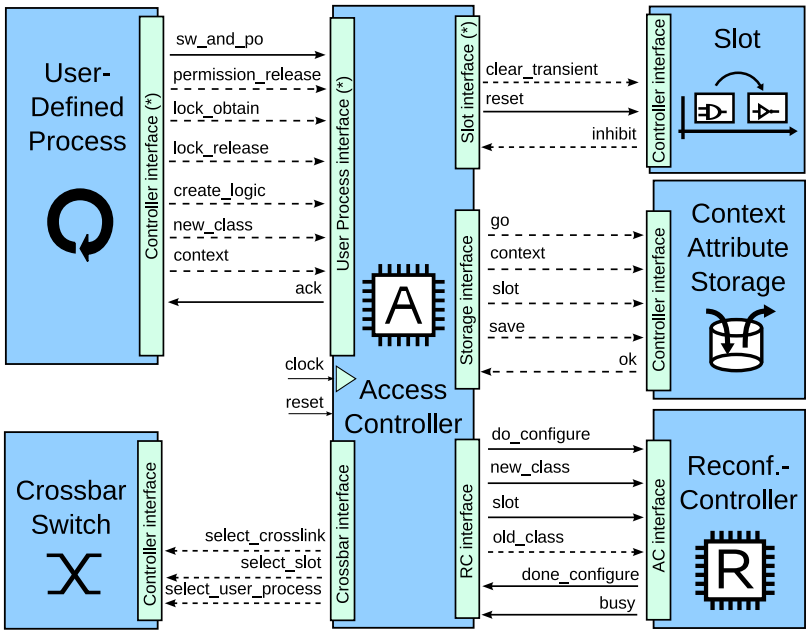

Figure 6.3: Block diagram: Access controller

Figure 6.3 shows the complete set of ports for an access controller. In case of a single user-defined process only, no permission handling is done and the signal `permission_release` is omitted. If no context locking is performed the appropriate signals are omitted. The same applies to the creation of classes.

Communication with User-Defined Process. Figure 6.4 shows a request by an user-defined process. The request includes access permission and a change in the dynamic

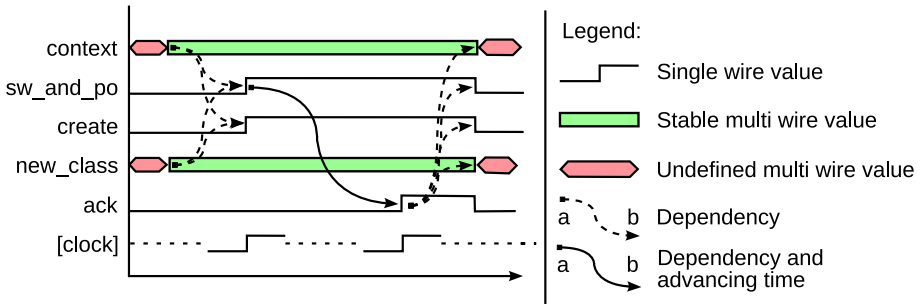

Figure 6.4: Timing diagram: Request and create

type of a context. The signal `switch_and_permission_obtain` (abbreviated `sw_and_po` in the diagrams) indicates that the user-defined process demands an object switch, as well as the permission to access the slot (to invoke methods). In case of a single user-defined process only, this signal is interpreted as a switch request only, since

permission handling is superfluous in case of a single client for a service. The `create` signal indicates that the runtime type of a context shall be changed. The access controller samples these values at a rising clock edge. At that instant, the auxiliary multi-bit signals `context` and `new_class` must be stable. Both are driven by the user-defined process. `context` transmits which context the access controller shall switch to. A special value is permitted to mark an anonymous access. The `new_class` signal indicates which target class is desired for that context. If `create` is low, the signal is ignored. In this example, the class switch was combined with obtaining an access. This is not necessary but saves some clock cycles. For both change in the runtime type (`create` signal) and context switch (or permission handling), the `context` signal indicates, on which context to perform the operation.

Figure 6.5 shows the protocol as a DFA. The inputs and outputs are signal levels on control signals. An upper case letter indicates driving a high value (all signals are active high) and a lower case letter a low value (inactive signal).

The input set is $I := \{p, P, c, C, r, R\}$ with P and p being the values of the `switch_and_permission_obtain` signal, $\{C, c\}$ the `create` signal values, and $\{R, r\}$ the `permission_release` signal values. The access controller drives the output set $O := \{A, a\}$ for `ack`. The initial state is `IDLE`. The user-defined pro-

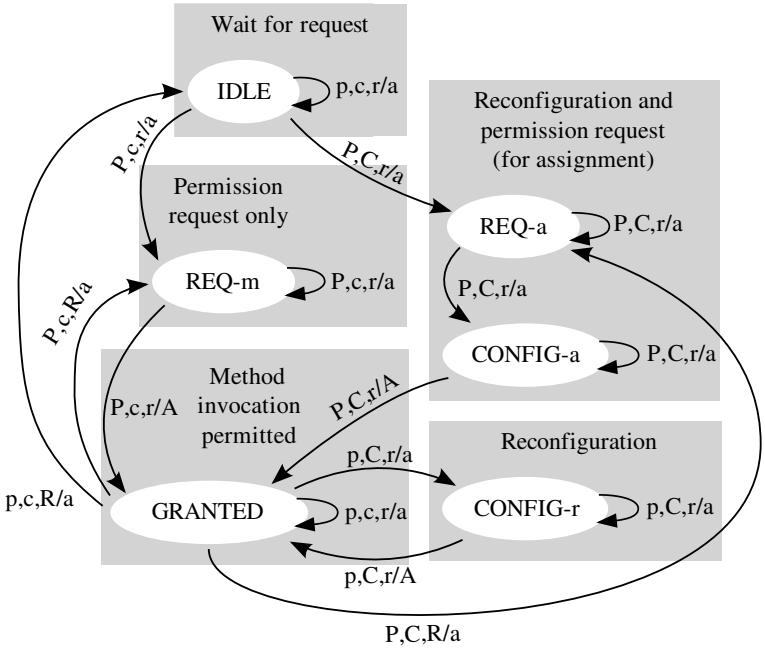

Figure 6.5: DFA: Protocol for access controller and user-defined process

cess may initiate a sequence of transitions in states `IDLE` and `GRANTED`. In these two states, the access controller is waiting for new requests to be serviced. In all other states, the input signals may not be modified.

In the `GRANTED` state, the user-defined process holds access permission for a context. To reach the `GRANTED` state, the user-defined process may request access permission by raising P. Additionally, it may also set C to high, requesting a context class switch as well. Depending on C, the states $REQ - m$ and $REQ - a$ are entered. In REQ-states the scheduling is done. Servicing the request may be postponed by the scheduler. Then, the DFA remains in the respective REQ state. If $REQ - a$ was entered and the request was selected to be serviced, a reconfiguration *may* occur, if the target slot does not have the correct configuration. To illustrate this, a separate state $CONFIG - a$ was introduced. In either case ($REQ - a$ or $REQ - m$), state saving

and restauration, placement calculation and other operations are done. They are not reflected in the DFA to keep it simple. The user-process is not aware about the progress of servicing its request until the transition to `GRANTED` is done. The access controller writes A for exactly one cycle. In the following cycle, the user-defined process must take back its P and C signals.

If the signals P and C are not at low level in the next clock cycle, this represents another request. If the access permission is returned by setting R, the protocol returns to `IDLE` state. If C (request for context object switch) is set while being in `GRANTED`, the object switch is requested for the context for which the user-defined process currently holds the access permission. For an object switch on the current context, the protocol enters `CONFIG-r`. When returning to `GRANTED` the successful completion of the request is indicated by setting A for a single cycle again.

A user-defined process is never being granted access to more than one context. If the context is to be switched quickly, the user-defined process may simultaneously return the old access permission (by setting R) and requesting a new one. These actions are shown by the edges from `GRANTED` to `REQ-a` and `REQ-m` respectively. The only exception is given, when it requests a class switch for the context it was just given access permission.

The process of locking and unlocking contexts is shown in Figure 6.6. Locking and unlocking of contexts is almost completely independent from the graph shown above. The only interdependency is that the multi-bit signal for the context number (named `CONTEXT` in the timing diagram) is re-used for these operations. As a consequence, the user-defined may not simultaneously request actions on one context while locking or unlocking another one. Both lock handling and returning of permissions can be done

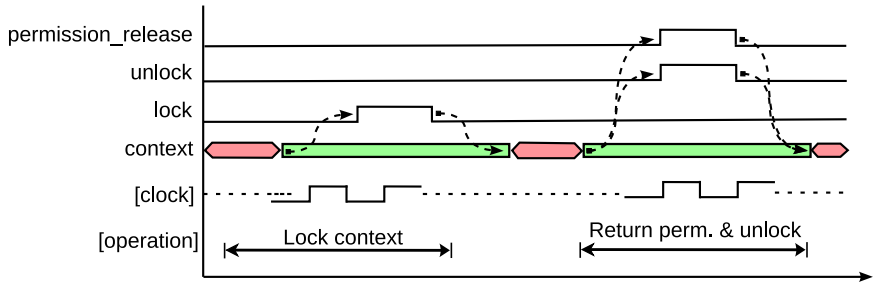

Figure 6.6: Timing diagram: Lock and simultaneous unlock and permission release

without waiting for an acknowledgment by the access controller. The user-defined process needs to set `lock`, `unlock` or `permission_release` to high for a single clock cycle. The `context` signal is re-used to refer to a context. As shown in the second half of the diagram, locking or unlocking may be done concurrently to returning access permission.

Communication with Slots. The access controller is also connected to the slot, although it does not invoke any methods on it. However, it resets the slot during system reset, and directly after each context runtime type change requested by a user-defined process. For the reset signal, it is not relevant whether a reconfiguration was performed or not. If the access controller omits a reconfiguration (because the requested logic is already in place), it performs a slot reset only. This reset is a single-cycle operation.

Communication with Crossbar. Once the user-defined process has been given a grant, it may access the slot. In case of a multi-slot reconfigurable object or multiple user-defined processes using the slots, a crossbar is inserted. The crossbar is transparent

to both slot and user-defined process. The access controller ensures that these communication partners are properly connected. This is done using the `select_slot` and `select_user_process` signals. These are sufficient, as long as only one slot or or one user-defined process is present. Then, both signals simply control multiplexer inputs.

The `select_crosslink` signal is added, if both multiple slots and multiple user-defined processes are present. Normally, it is driven by the access controller with an special value indicating that there is no action to perform. The crossbar is equipped with a process reading this signal. It manages a set of crosslinks (see Section 4.4.5) and watches the `select_crosslink` signal for which link to modify. If a valid crosslink number is given, it reads the `select_slot` and `select_user_process` to modify that link. The access controller is responsible for immediately returning `select_crosslink` to the idle value after a single clock cycle. Having a process inside the crossbar delays all link switches by one clock cycle compared to the fully combinatorial version of the crossbar. However, this delay doesn't have an influence on the circuits functionality. The access controller adjusts the crossbar concurrently to granting access permission to the user-defined process. The first communication between user-defined process and slot will not occur before the following clock cycle. Figure

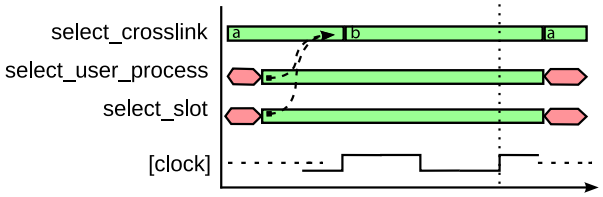

Figure 6.7: Timing diagram: Crossbar

6.7 describes a modification in the crossbar. Initially, the `select_crosslink` indicates that there is no action (a). Then, a switch of crosslink number "b" is to be done. Both `select_user_process` and `select_slot` are driven with the identification of the user process that is to be connected to a specific slot. Then, the signal `select_crosslink` is driven with "b" before a rising clock edge to initiate the modification. One clock cycle later (indicated by the dotted vertical line) the user-defined process and crossbar are connected. Finally, the access controller resets `select_crosslink` to "a".

Communication with Slot. In case of transient attributes, the access controller may also request resetting the transient attributes to their default values. This is done after restoring a context's dynamic state by using the `clear_transient` signal.

In case of inner locks being present in any class potentially being configured to the reconfigurable object, the slot provides a `inhibit` signal. While this is true, the inner lock inhibits a reconfiguration.

Communication with the Context State Storage. The access controller may decide to save or restore a context's dynamic state. This is a task being performed by the context attribute storage. An access controller requests a save or restore process from the storage component by asserting the `go` signal. The signals `context`, `slot` and `save` are used to describe the task and must remain stable through the operation. `save` is true, if a context is to be saved and false when restoring a context. The access controller awaits `ok` to be set for one single clock cycle and deasserts `go` in the following clock cycle.

The reconfiguration controller interface is used if the access controller decides to perform a reconfiguration. Similar to context saving and restoring, it asserts the signal `do_configure` and awaits `done_configure` being set for one cycle. The `new_class` and `slot` signals describe, which logic to configure where. `old_class` is optional. If differential bitstreams are available, this signal is transmitted to inform the reconfiguration controller about the current state of the slot. The reconfiguration controller might use this information to pick a configuration bitstream that is faster to apply. During reconfiguration, the controller asserts its `busy` signal. It is forwarded to the scheduler inside the access controller to allow optimized decisions.

6.2.2 Crossbar

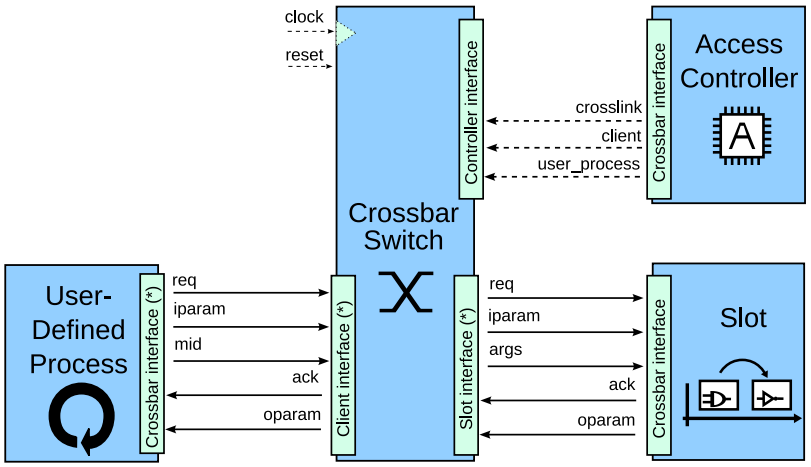

Figure 6.8: Block diagram: Crossbar

The crossbar connects slots and user-defined processes. It is a bi-directional multiplexer structure managed by the access controller. In case the `select_crosslink` signal is present, a process is embedded into the component. In that case, the crossbar receives both `clock` and `reset` ports. A detailed description of its interaction with the access controller is given in Section 6.2.1.

6.2.3 Slot

The slot module is the instance being reconfigured and its communication is shown in Figure 6.9. Its communication partners, namely crossbar, access controller and context attribute storage ignore signals from the slot module during slot reconfiguration. The slot module may drive arbitrary values on its outgoing signals during reconfiguration. For a description of the access controller interface, see Section 6.2.1. Method executions are initiated by user-defined processes. The `req` signal indicates an incoming request for method execution. As Figure 6.10 shows, the slot may rely on the signals `mid` and `iparam` being stable when sampling `req` at a rising clock edge. They remain stable until `req` is deasserted. Once the slot finished executing a method, it asserts `ack` and provides the return value as well as copy-in-copy-out values (for reference parameters) in `oparam`. In the following cycle, both `req` and `ack` must be deasserted. If `req` remains asserted, the slot assumes the beginning of the next method invocation, which is legal. This way methods, which can be executed in a single clock cycle can be issued every two clock cycles.

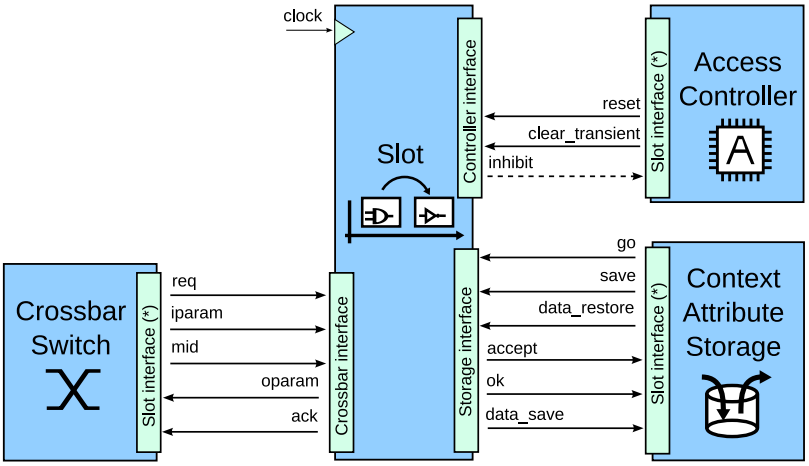

Figure 6.9: Block diagram: Slot

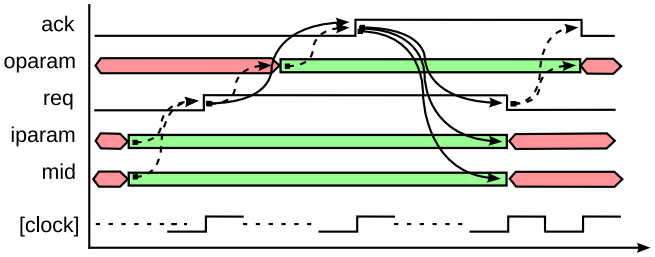

Figure 6.10: Timing diagram: Method invocation

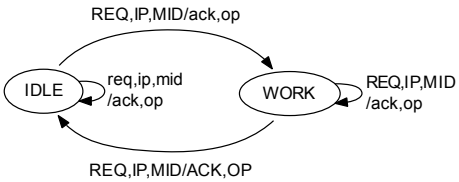

Figure 6.11: DFA: Protocol for user-defined process and slot

Figure 6.11 shows the communication between a user-defined process and a slot. The inputs $I := \{REQ, req, IP, ip, MID, mid\}$ are driven by the user-defined process. $\{REQ, req\}$ are request control signal values, the uppercase word reflects an active signal. IP refers to input parameters driven stable, while ip means undefined values for these wires. The same applies to the method identification multi-bit signal represented by $\{MID, mid\}$. The output set $O := \{ACK, ack, OP, op\}$ refers to the acknowledge control signal and the multi-bit signal for the output parameters ($\{OP, op\}$).

Methods are initiated by driving REQ along with stable data wires for input parameters and method identification. Method executions start immediately and the protocol enters the state `WORK`. The input signals must remain stable until the slot acknowledges the completion of method execution using ACK. This is the transition back to the state `IDLE`. In the following cycle the acknowledgment is taken back. If the request wire is also taken back, then the system remains in the idle state. Otherwise, another method execution is initiated.

The interface to the context attribute storage is described in Section 6.2.4

6.2.4 Context Attribute Storage

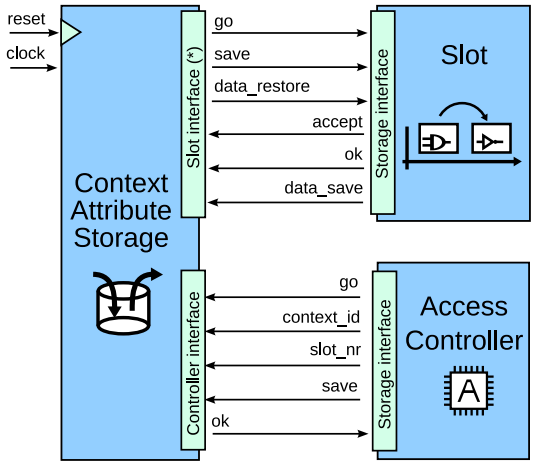

Figure 6.12: Block diagram: Context attribute storage

Figure 6.12 shows the ports of a context attribute storage. Its access controller interface was described in Section 6.2.1.

The context attribute storage saves and restores a slot's dynamic state. The storage contains a memory, which is logically split into individual sections for each permanent context. Each section is sized to accept the largest possible state for the given permanent context. The sizes may differ depending on the context's interface class. Upon saving and restoring a context's state, the storage component has no information about the amount of datawords to be transferred. The slot, however, knows this value, since it depends on the runtime class it currently implements.

The protocol works as illustrated in Figure 6.13. The inputs $I := \{GO, go, S, s\}$ are driven by the context attribute storage, while the outputs $O := \{OK, ok, A, a\}$ are driven by the slot. GO and OK are control signals for handshaking. S indicates a save process and s a restauration process. During save, a data multi-bit data signal is driven by the slot and another data signal is driven by the context attribute storage during restauration. The edges marked with an asterisk (*) indicate, when a new value is to be driven. It needs to remain stable until an edge towards the `IDLE` state is taken.

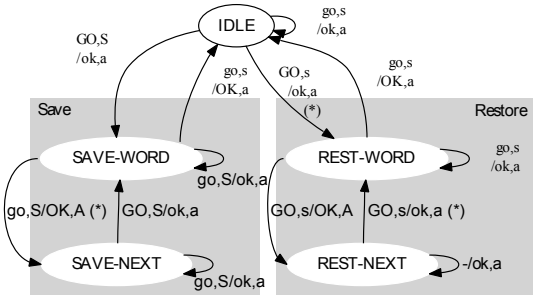

Figure 6.13: DFA: Protocol for context attribute storage and slot

The `IDLE` state is left when the context attribute storage initiates a process by writing GO along with S or s.

The data is copied word-wise. Upon restoring data, each GO (held at high level for exactly one cycle) is answered with an OK by the slot. While valid words are transmitted, the slot sets A (for *accept*) along with OK. This is the transition from `SAVE-WORD` into `SAVE-NEXT`. Both acknowledgment signals are also taken back after one cycle. While in `SAVE-NEXT`, the next word may be transmitted, returning into `SAVE-WORD`.

Since the slot is configured with the appropriate logic at the time of copying data, the amount of data to be copied is known by the slot as a built-in value. Once a data-word is written which exceeds the memory to be transferred, the OK signal is set with a (and not A), therefore rejecting the word. This ends the transmission, returning from `SAVE-WORD` to `IDLE`.

The state restauration works very similar. The main difference is that the data words are written by the context attribute storage and read by the slot. This process is initiated by setting s along with GO.

This protocol has two important properties: First, the context attribute storage may wait arbitrary times until it sets `go` for each data word. It allows slower implementations of storage devices for its internal memory. It may well take multiple cycles to load or store a data word. Second, the context attribute storage does not need to know the amount of data words to be transferred. It copies data until `accept` is not set. It is safe to copy undefined data after writing all valid data to the slot, since it will be discarded anyway. The same applies to the state saving process: The context attribute storage will not receive invalid data since the slot will terminate the transmission.

6.2.5 User-Defined Process

Figure 6.14 shows the communication ports used by user-defined processes. To be precise, the ports shown in the figure are added to each module containing a process using a reconfigurable object and once per object. The communication protocols are described in Sections 6.2.1 and 6.2.3.

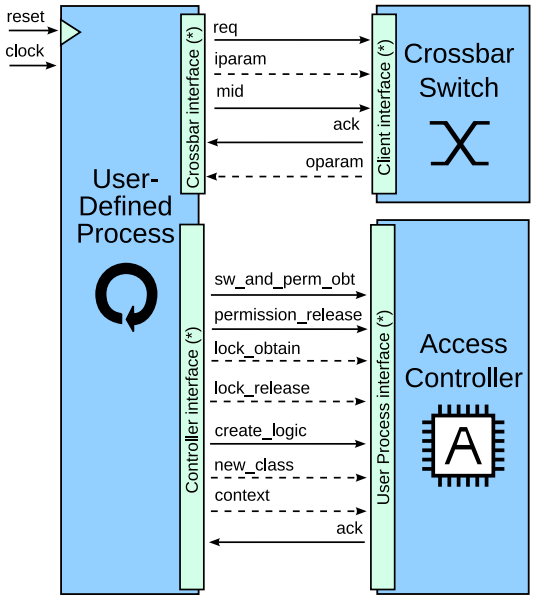

Figure 6.14: Block diagram: User-defined process

6.2.6 Reconfiguration Controller

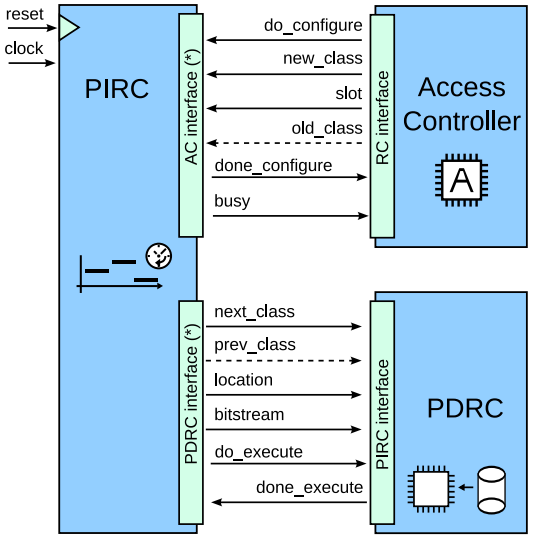

Figure 6.15: Block diagram: Reconfiguration controller

Figure 6.15 shows the **p**latform **i**ndepent part of the **r**econfiguration **c**ontroller (PIRC). The PIRC unconditionally executes reconfiguration requests from all access controllers on one FPGA device. The protocol between the access controller and the PIRC is described in Section 6.2.1. The PIRC contains a scheduler, if more than one access controller is connected. The scheduler decides about the order in which to serve the reconfiguration requests.

Once a scheduling decision was made, the **p**latform **d**ependent part is instructed to execute the reconfiguration unconditionally. Since the PDRC does not know about the

logical hierarchy of reconfigurable areas of slots associated to different reconfigurable objects, the slot identifications are translated into a `location` information. This is a flat numbering of all reconfigurable areas.

The PIRC specifies the bitstream to be configured in two ways: First, it provides it as an identification number for the requested class (signal `new_class`) along with an location number (signal `location`). The PDRC may then use these two signals in case of position-independent configuration bitstreams being available. Additionally, the signal `prev_class` may be used to use bitstreams that are both position independent and differential. Secondly, the PIRC provides a translated bitstream identification, in case of just position-dependent bitstreams being available. During synthesis, *fossy* generates a description file containing the bitstream tables.

A reconfiguration is initiated by rising `do_execute`. The PIRC waits for the signal `done_execute` to rise and both signals are then deasserted in the following clock cycle.

6.3 Board Support Package

The *fossy* tool is platform independent. Therefore, it is only able to generate those part of the reconfiguration controller that is not platform specific. A complementary, platform specific part needs to be provided by a *board support package* (BSP). It is to be implemented manually for a given target platform. The interface between the platform specific and platform independent part is designed as a loose coupling. This eases re-use of board support packages for different applications.

In the ANDRES project [1] a board support package for Xilinx ML401 and ML501 evaluation boards was implemented. The BSP provides the platform dependent reconfiguration controller (PDRC) for OSSS+R. It interfaces with the platform independent part of the design and executes reconfiguration requests. Additionally, an utility script to assist with Xilinx' Early Access Partial Reconfiguration (EAPR) Flow (see Section 7.1) is provided.

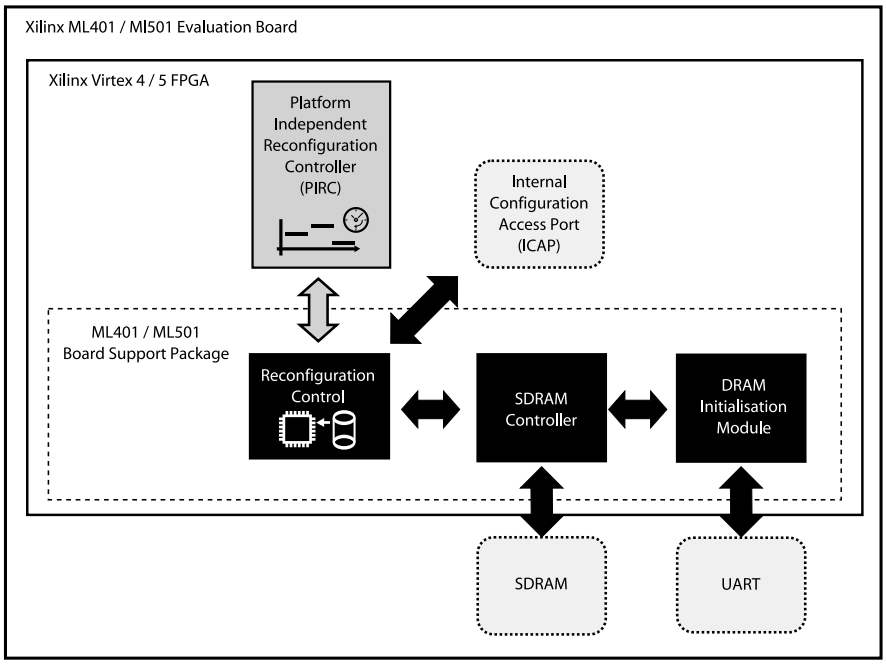

Figure 6.16: Block diagram for ML401/ML501 board support package

Figure 6.16 shows a part of a design on a ML401 or ML501 evaluation board. The components of the board support package are shown in black color with white text. The board support package implements the platform dependent part of the reconfiguration controller. It consists of three modules:

1. SDR-DRAM initialization module
2. SDR-DRAM controller module
3. Reconfiguration control module

Among the resources provided by Xilinx ML401 and ML501 evaluation boards a FPGA, SDR-DRAM and a serial UART interface are used for the board support packages. The SRAM is used to store partial configuration data. It is interfaced by the SDR-DRAM controller module. First, the memory layout for a given benchmark is calculated on a host PC. The partial bitstreams are assembled to a memory image and stored in a file.

The benchmark is started as follows. First, a full configuration is downloaded to the FPGA using Xilinx' iMPACT tool. Then, the initialization module takes control. It receives the pre-calculated memory image using the UART interface. The data is handed over to the SDR-DRAM controller module, which stores it inside the on-board memory. This makes the system operational. When the platform independent reconfiguration controller (PIRC) part requests a reconfiguration, the PDRC serves this request. It fetches the configuration data via the RAM controller module from DRAM and feeds it into the FPGA's internal configuration access port (ICAP). After finishing, it acknowledges the request to PIRC. Figure 7.3 shows the components of the PDRC as boxes with black background and white text.

6.4 Chapter summary

This chapter presented the synthesis of reconfigurable objects. In the first part, the steps performed by the *fossy* tool are explained. Only those features are described, which are implemented in *fossy* . The second part explains a set of register transfer level components that can be used as building-blocks to implement an OSSS+R model. Here, the not yet automated parts are included as well. Finally, an example board support package is presented, too.

Chapter 7

Experiments and Evaluation

There are multiple possible applications of dynamic partial reconfiguration (DPR). Section 1.3 introduced three common goals of DPR. The most prominent is the increase of resource utilization. Parts of the design are adapted at runtime to implement various operations. This eliminates the need to implement circuitry to service all functionality statically. The area demand is reduced in comparison to fully static solutions, thus increasing the resource utilization. This is the goal OSSS+R was designed to help achieving.

The potential for effectively applying DPR heavily depends on the application to be implemented. Section 1.3 contains a set of application properties that can be expected to be necessary. There are several case-studies showing that the circuit area can be reduced by using DPR for some applications. Examples can be found in [70, 80, 35].

This chapter presents the benchmarks used to evaluate the proposed approach. As Table 3.1 shows, OSSS+R is one of the approaches providing accuracy on the clock cycle level when comparing the abstract model with the final implementation. The simulation accuracy is verified by checking the simulation timing against the OSSS+R specification. This is done using testcases. One example testcase is presented in Appendix A. The functional equivalence of the OSSS+R simulation model and its corresponding register transfer level simulation model (as generated by *fossy*) is checked using different testcases. To give an example, Appendix B contains a benchmark called `dispatch`. It checks the functional correctness of both simulation models.

An overview on the tool flow is given in Section 4.1. Extending that explanation, the vendor specific steps are shown in this chapter. Although the methodology is platform independent, implementing the synthesis results on a specific FPGA device does require adaptions, which do depend on the device vendor's back-end tools. The experiments were done using Xilinx Virtex 4 and Virtex 5 devices using the Xilinx Early Access Partial Reconfiguration (EAPR) flow.

Next, an audio-generation benchmark is introduced, which was initially specified as a C++ model. All steps of the tool flow were performed, resulting in an implementation running on both Xilinx ML-401 (Virtex 4) and ML-501 (Virtex 5) evaluation boards.

Then a second benchmark using a cyclic redundancy check (CRC) is shown. Although the core algorithm is not large enough to motivate DPR, the benchmark is quite flexible. It can be tuned to demonstrate the area impact of various design decisions.

> Note: Neither of the two benchmarks is designed to demonstrate potential area-saving abilities when using dynamic partial reconfiguration. Implementing another case-study of such size and complexity would be time-consuming and exceed the time-frame available for this thesis.

Instead, the benchmarks were selected to demonstrate the abilities of this approach: The audio-generation benchmark shows the seamlessness of the flow, ranging from an

abstract specification to an implementation on a FPGA evaluation board. The second benchmark is more artificial and crafted to show the influence of design decisions on the size of the resulting circuit. It was designed to allow a large variety of implementations. From a closer look at the figures of these two benchmarks it will become clear that OSSS+R will most likely exploit the area-saving potential, if the application is suitable for DPR.

Finally, the approach is evaluated with respect to the goals as shown in Section 1.3.

7.1 Implementation Using Xilinx EAPR Tool Flow

The last step of the vendor independent part of the design flow is the generation of an implementation model using the *fossy* tool. The output are VHDL files containing the design as a synthesizable register-transfer level model.

If *fossy* is executed with the `-multipleOutputFiles` option, it generates a set of output files, one for each VHDL entity. These can be grouped into a set of files for the static (non-reconfigurable) part of the design and further ones for each implementation variant of each dynamic part.

These are to be mapped to the static and dynamic regions of the FPGA device. In OSSS+R terminology, the dynamic parts of a design are called *slots*. They are assumed to be of fixed size and provide different variants that are mutually exclusive during runtime. In the terminology of Xilinx' EAPR tool flow, these refer to *partial reconfigurable regions* (PRRs) containing mutually exclusive *partial reconfigurable modules* (PRMs).

The smallest configurable unit is a *frame*, which has a varying size depending on the device family. In earlier Xilinx architectures (e.g. Virtex-2), frames need to span the whole height of a device. More recent architectures like Virtex-4 and Virtex-5 allow frames of 16 or 20 logic blocks (CLBs) for frames, respectively. Each CLB consists of four slices, which are arranged in two rows and two columns. The FPGA area represents a grid of slices. PRRs may start at even slice columns and rows and end at odd slice columns and rows. Consequently, it is permitted that a PRR affects only parts of frames. In this case, some parts of a frame are not modified (but contribute to the configuration bitstream size).

The Xilinx EAPR flow [144] requires the design to have some properties. First, it needs to be ensured that the top level design instance does not directly include processes. Instead, all of them need to be encapsuled in submodules. The top-level module may only contain submodules, I/O components, signal declarations, buffers, clock managers, and so-called *bus macros*. On the other hand, submodules for reconfiguration may not contain certain components (like buffers) and others require special attendance (I/O components). If the design does not comply to these rules, it needs to be modified manually.

The *bus macros* are used to implement the interface between the static and dynamic parts of the design. All signals crossing the border of a PRR must go through so-called *bus macros*. Their ports are used to name endpoints for connections to be drawn by the router. Therefore, they are an artifact required by of the used routing software.

If the HDL sources meet all the requirements, the steps of the EAPR flow may be executed, as shown in Figure 7.1. First, the HDL sources are compiled to netlist, for example using the Xilinx *XST* tool.

Second, a constraint file needs to be created. For non-reconfigurable designs, the placement of Xilinx specific primitives like I/O-interfaces, digital clock managers, block RAMs, and other components. Also, the location of bus-macros is specified here, as well as partial reconfigurable regions (PRMs).

Once this file is ready, mapping and placement is done. First, the static design is implemented. It is possible that static design parts are placed inside the PRRs. These are marked as occupied and an exclusion file is generated. By including this exclusion

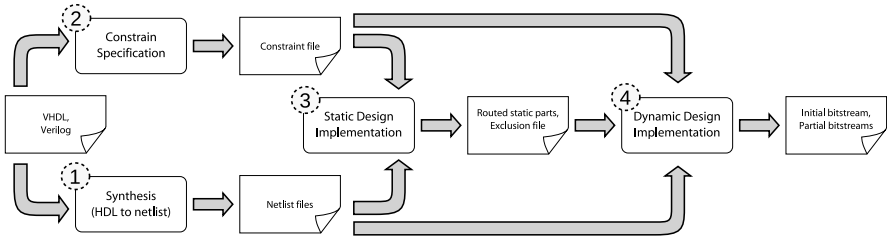

Figure 7.1: Xilinx EAPR flow (simplified)

file, the PRMs can be translated inside the PRRs without resource conflicts between the static and dynamic design parts. This is shown in step four in Figure 7.1.

For the waveform generator, the EAPR flow was partially automated by a set of scripts. It was created within the ANDRES project[1]. At the University of Paderborn, a tool named Part-E[39, 40] was developed. It allows creation of top-level modules suitable for the EAPR flow, including PRM layout, bus macro insertion and, constraint file generation. Such a generated top-level design file could be used if modifying the top-level design file that is generated by *fossy* manually would take more time.

7.2 Benchmark: Waveform Generator

This first benchmark is intended to show the feasibility of the flow, starting from a C/C++ algorithmic specification ranging down to a working implementation on a Xilinx evaluation FPGA board.

C++ Implementation

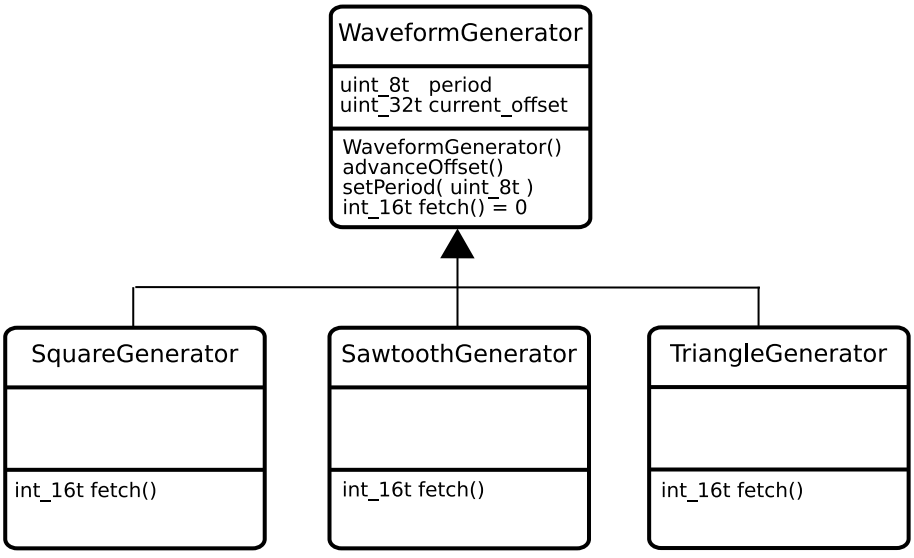

Figure 7.2: Class diagram: Waveform generators

The application used for this benchmark generates an audio signal of 48 kHz sample rate and 16 bits depth. It uses a set of waveform generator classes inherited from a

common base class providing their interface. Figure 7.2 shows the class diagram. During runtime, exactly one object is used to generate a waveform. The implementation uses a polymorphic pointer to dynamically create and destruct objects. This way, the pointer refers to different waveform generator algorithms. All algorithms can be tuned to a different period, allowing to alter the frequency of the generated waveform.

A different functionality generates an envelope. It is tunable with respect to its attack, decay and release times, as well as its sustain level. Calling a `keystroke()` method, new strokes of a user-defined length can be initiated.

Each sample of the current waveform is then amplified by the envelope. Finally, the sound is filtered using a low pass filter, a further amplification swelling up and down in a periodical loop or, left untouched. The loop amplification also has a adjustable period. Like the waveform generator, the filter is done using polymorphism by replacing filter objects at runtime.

The resulting audio file is then written to a file as a Sun Microsystems `.au` audio file for playback on a PC. The following code fragment shows the generation of samples:

```
1  for (unsigned int second = 0; second < 10; ++second) {
2    // "eg" is the envelope generator object
3    eg.keystroke( ONE_SECOND >> 2 );
4
5    for (unsigned int sample_nr = 0;
6         sample_nr < ONE_SECOND;
7         ++sample_nr)
8    {
9      sample = filter.process(amplify(wave_gen.fetch(),
10                                     eg.fetch()));
11
12     // next: writing sample to file
13     ...
14   }
15 }
```

This is the main routine of the C++ model. The sample generation (lines 3 to 10) is going to be implemented in the hardware design while the file I/O and is moved to the testbench.

OSSS+R Implementation

In a first step, the C++ software model was manually transformed into a hardware description. Waveform generator, envelope generator, amplification, and filtering was moved to concurrent, clocked processes (`SC_CTHREAD`s). They read and write signals carrying 16-bit samples at 48 kHz frequency. The filter classes and waveform generator classes were annotated as shown in Section 5.1. The waveform generator and filter were chosen to be reconfigurable. Reconfigurable objects replace the pointers. Method invocations remain untouched, while assignments are slightly modified: `wg = new wave::SquareGenerator()` is being replaced by the statement `wg = wave::SquareGenerator()`, omitting `new`. Also, `delete` calls were removed.

Then, inputs from the testbench were added to request the changes of the waveform or filter type. The following code sequence shows the process performing the waveform generation.

```
1 void WaveGenModule::work()
2 {
3   sample_out.write( wave::NEUTRAL_SAMPLE );
4   wait(); // Reset code ends here
```

```
5
6   // 0 = SquareGenerator
7   // 1 = SawtoothGenerator
8   // 2 = TriangleGenerator
9   sc_uint< 3 > current_generator = 4; // Force a switch
10
11  while (true)
12  {
13    // Adapt generator
14    if (generator.read().to_int() == 0) {
15      wg = wave::SquareGenerator();    current_generator = 0;
16    } else if (generator.read().to_int() == 1) {
17      wg = wave::SawtoothGenerator(); current_generator = 1;
18    } else if (generator.read().to_int() == 2) {
19      wg = wave::TriangleGenerator(); current_generator = 2;
20    } else wait();
21
22    // Loop while no generator switch requested
23    while (current_generator == generator.read())
24    {
25      if (period.read().to_int() == 0)      wg->setPeriod( 6 );
26      else if (period.read().to_int() == 1) wg->setPeriod( 7 );
27      else if (period.read().to_int() == 2) wg->setPeriod( 8 );
28      else if (period.read().to_int() == 3) wg->setPeriod( 9 );
29      else wait();
30
31      // Generate a sample
32      wave::sample next_sample = wg->fetch();
33
34      // Write next sample (synchronous to 48kHz)
35      do
36      {
37        wait();
38      } while (false == sync.read());
39      sample_out.write( next_sample );
40      do
41      {
42        wait();
43      } while (true == sync.read());
44    } // while same generator
45  } // while true
46 }
```

In line 9, a state variable `current_generator` for the active waveform generator algorithm is defined. It is initialized with an invalid value, which causes an initial change to a valid value as specified by the `generator` input port (which is written by the test bench).

The `while (true)` loop body starts with an adaption of the current generator. According to the instruction given by the testbench (input port `generator`) the algorithm is adapted. This makes use of the feature of OSSS+R that redundant reconfigurations are automatically omitted. If the algorithm matches, the assignment does not cause a reconfiguration. It resets the algorithm to its default values only.

Lines 23 to 44 contain a loop that iterates, as long as the requested generator algorithm matches the current one. If a mismatch is detected, the loop is left and the algorithm is adapted, then the while-loop is entered again.

The first action inside the loop body is a sequence of statements adjusting the waveform period. This is a fast method call with only a few clock cycles duration. Next, in line 32 a new sample is fetched from the reconfigurable object `wg`.

Lines 35 to 43 write the fresh sample, as soon as an rising edge of the `sync` signal is detected.

Next, the module declaration is shown. It contains the declaration of the reconfigurable object instead of the polymorphic pointer.

```
1  SC_MODULE( WaveGenModule )
2  {
3    public:
4    sc_in< bool > clock;
5    sc_in< bool > sync;
6
7    sc_in< bool > reset;
8    sc_in< sc_uint<2> > period;
9    sc_in< sc_uint<2> > generator;
10
11   sc_out< wave::sample > sample_out;
12
13   osss::osss_recon< wave::WaveformGenerator > wg;
14   //In C++ code: WaveformGenerator * wg;
15
16   void work();
17
18   SC_CTOR( WaveGenModule );
19 };
```

The only OSSS+R specific part of this listing can be found in line 13. It shows the declaration of the reconfigurable object `wg` with the user-defined interface type `wave::WaveformGenerator`. Line 14 shows its origins from the C++ model.

The next listing shows the module's constructor. Note that the reconfigurable object is given a name (line 3), the process `work` is declared to use `wg` (line 7), and its clock and reset ports are connected (lines 9 and 10).

```
1  WaveGenModule::WaveGenModule( sc_module_name name_ )
2          : sc_module( name_ ),
3            wg("wg")
4  {
5    SC_CTHREAD( work, clock.pos() );
6    reset_signal_is(reset, true);
7    uses( wg );
8
9    wg.reset_port( reset );
10   wg.clock_port( clock );
11 }
```

The modifications in the filter module are similar. Next, a prototyping platform definition and declaration was inserted:

```
1  // Device type and instance
2  osss::osss_device_type my_device_type("ML401");
3  osss::osss_device my_device (my_device_type,
4                               sc_module_name("my_device"));
5  // Device port bindings (for reconfiguration control)
6  my_device.clock_port(s_fast_clock);
7  my_device.reset_port(s_reset);
```

Then, the reconfigurable modules are bound to the device instance:

```
1 // RO placements
2 top.producer.wgm.wg( my_device );
3 top.producer.fm.filter( my_device );
```

And finally, estimated reconfiguration and context copying times are added. These may be replaced, once more reliable estimates are available. The code excerpt shows the declaration for one class:

```
1 OSSS_DECLARE_TIME(my_device_type,
2                    wave::SquareGenerator,
3                    sc_time(2, SC_US),
4                    sc_time(2, SC_US));
```

The transformed benchmark was simulated using the OSSS+R modeling and simulation library and the OSCI SystemC 2.2 implementation. It simulates reconfiguration for the waveform generator and filter components.

Up to this point, it still wrote the generated waveform into a file on the simulation host. Since file I/O is not synthesizable, this was replaced with a AC'97 controller [7] written in SystemC for this benchmark. The testbench was equipped with a dummy AC'97 codec receiving the samples and dumping them into a file.

After simulation, it can be synthesized using the *fossy* tool. As explained in Section 4.1, there are two possible models that can be generated. One of these is a register-transfer level model. Figure 7.3 shows a block diagram of the generated model.

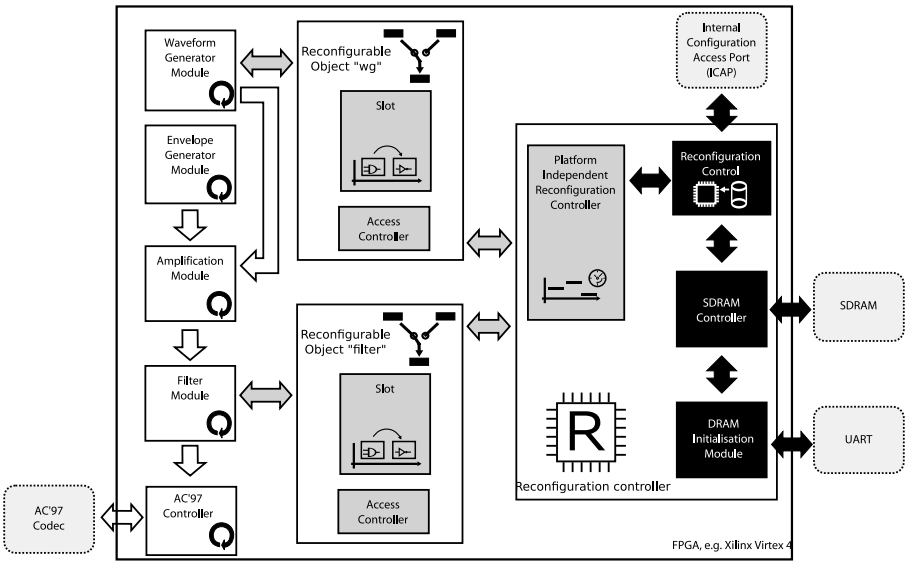

Figure 7.3: Block diagram: Waveform generator synthesis model

The white boxes show the components of the OSSS+R model as specified by the designer. Reconfigurable objects and device declaration are resolved by the synthesis tool into a set of replacement components, shown in grey color. Each reconfigurable object is replaced by a reconfigurable area (containing one slot only) and an access controller. All user-defined processes manipulating reconfigurable objects are touched as well. The modules containing these processes are equipped with additional ports for communication with the access controller and slot. Manipulation statements inside the user-defined processes are then replaced by an equivalent signal-level protocol. The access controllers are then connected to the platform independent part of the reconfiguration control unit. This unit performs scheduling of reconfiguration requests and

relies on the platform dependent part for their execution. The platform dependent part is shown in black. It is to be added from the board support package of the used prototyping platform. Inside the ANDRES project [1], board support packages for Xilinx ML-401 and ML-501 boards were developed and re-used here (see Section 6.3).

RT-Level Simulation Model

To validate the correctness of the gray parts and modifications of the user-defined processes, a simulation model was generated, too. This simulation model is identical to the synthesis model, except for the platform dependent reconfiguration part. It's block diagram is shown in Figure 7.4.

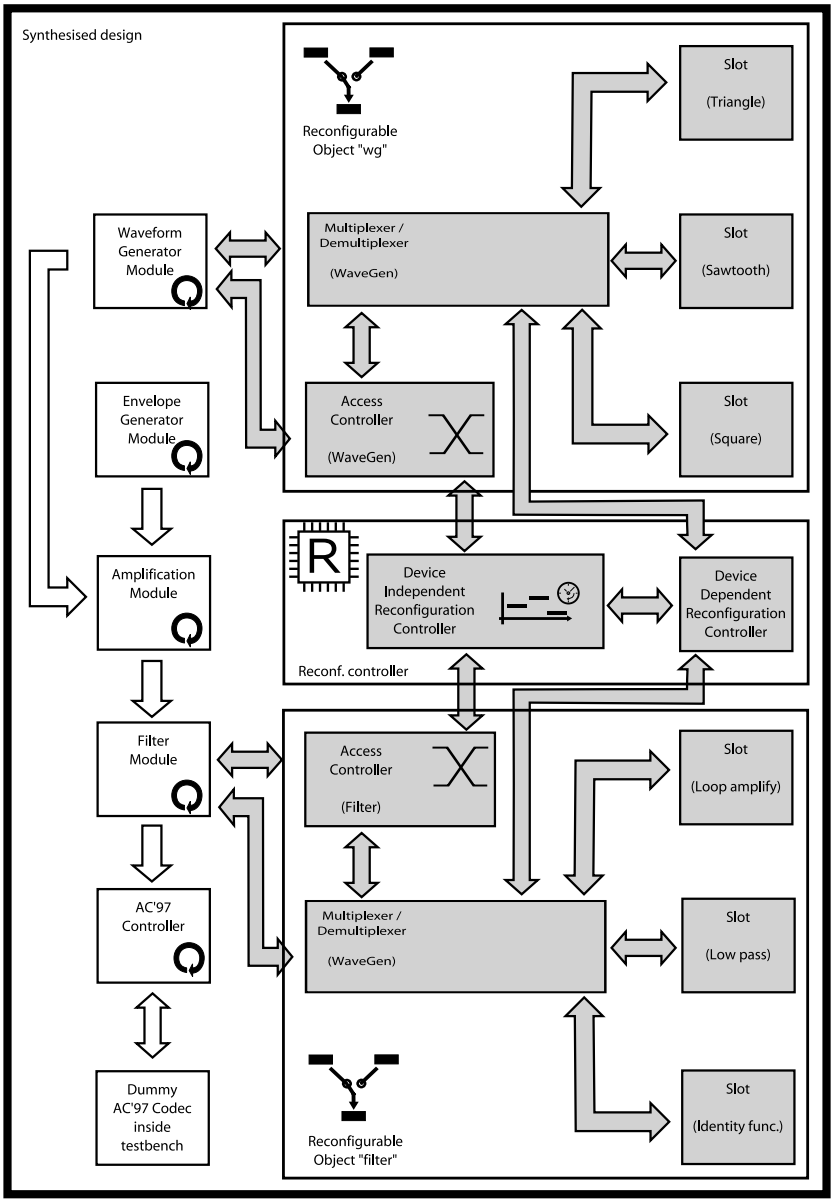

Figure 7.4: Block diagram: Waveform generator simulation model

To simulate the effect of dynamic reconfiguration, all variants of a reconfigurable

area are instantiated simultaneously. A multiplexer unit switches among them at the end of each simulated reconfiguration. The platform dependent part of the reconfiguration controller is also generated by *fossy* . It delays the switching of multiplexers and resets the slots to mimic the final system behavior. On the other hand, all access controllers, the platform independent reconfiguration controller, and all modifications inside user-defined process exactly match those of the synthesis model. The simulation model may be loaded into standard hardware simulators like Mentor Graphics ModelSim.

The functional correctness of this model was tested by a SystemC-VHDL co-simulation. The generated VHDL code was embedded into the same SystemC testbench as used by the OSSS+R model. In this testbench, the AC'97 codec is simulated. The audio samples received by the codec are written to a file and then compared to those of the OSSS+R model.

FPGA implementation

Table 7.1 shows the area utilization on a Xilinx Virtex-4 FPGA. The LX-25 device is the second-smallest of the Virtex 4 LX family with 21504 slices. The figures repre-

Design part	Look-up tables	Flip-flops	Slices	Percentage
Static modules	1977	1085	1278	11,89%
PRM filter base class	54	19	37	0,35%
PRM low pass	73	35	48	0,45%
PRM loop amplify	133	48	92	0,86%
PRM sawtooth	224	80	145	1,35%
PRM square	223	76	147	1,37%
PRM triangle	223	63	149	1,39%
Virtex 4 LX-25 total size	21504	21504	10752	(100%)

Table 7.1: Waveform generator benchmark: Resources after P&R

sent values after placement and routing. The utilization percentage is calculated based on slices. At that time, the individual modules of the static design part can no longer be identified. Using Xilinx PlanAhead tool, a breakdown of individual module use is possible, as shown in Table 7.2. It uses the mapped netlist files for its estimation. However, these values are less accurate than those available after placement and routing. This benchmark demonstrates that the complete flow from C/C++ specification down to an operational design on a FPGA prototyping platform is functional. Furthermore, it can be seen that the area overhead due to the use of the OSSS+R methodology is not excessive. The design requires less than 15% of the FPGA's logic resources. The benchmark itself is rather small. Since each reconfigurable area may be loaded with three alternatives each, and each alternative is considerably small itself, the benchmark is not suitable to demonstrate the benefit of partial reconfiguration itself. On the other hand, the benchmark shows that implementing self-reconfiguration using two reconfigurable areas is possible using OSSS+R without generating a large overhead. A discussion of acceptable overhead is given in Section 7.4.

To further investigate the area impact of OSSS+R statements, a second benchmark is analyzed next.

7.3 Benchmark: Cyclic Redundancy Check

Since, it is difficult to tell, which logic element in the final netlist originates from what high-level statement, a different evaluation approach is chosen.

Design part	Look-up tables	Flip-flops	Slices
PRMs, fossy-synthesized			
Filter base	35	19	11
Low pass	54	35	17
Loop amplify	128	48	40
Sawtooth	234	80	72
Square	244	63	75
Triangle	244	76	75
Static modules, fossy-synthesized			
AC'97 Controller	294	228	90
Debounce	18	10	6
PIRC	333	70	102
Producer	602	374	184
Static modules, board support package			
PDRC Initialize	464	235	142
PDRC Control	211	121	65
PDRC SRAM Control	33	86	27
Static total	1955	1124	616
FPGA total	21504	21504	10752

Table 7.2: Waveform generator benchmark: Resource breakdown

An artificial benchmark is used that implements a cyclic redundancy check (CRC). In its simplest form, a process reads input data from a stream and calculates the check bits. The input data also selects among a pre-defined set of polynomials to use. The benchmark implementation is done in a way that four dimensions of modifications are possible, allowing observing their impact. These are:

1. Increase of processes performing calculations.
2. Selection between exclusive computational resources or one shared resource (requiring arbitration).
3. Selection whether to use a entirely static or reconfigurable implementation.
4. Increase the set of polynomials.

Polymorphic Object Version. In the simplest form, the benchmark consists of one process in one module. The only process receives a data stream from testbench and calculates the CRC checksum using a polymorphic object. Increasing the number of user-defined processes just means replicating this circuit. Each user-defined process has its exclusive computational resource.

Shared Object Version. In a second version, a shared resource for computation is used. Figure 7.5 shows such a situation. Without dynamic partial reconfiguration, this resource is implemented using a shared object. Since, shared objects are not polymorphic themselves, a polymorphic object is wrapped inside the used object. The shared object is instantiated with an auxiliary class that forwards all method invocations to a polymorphic object that is instantiated as a data member. Since shared objects also lack support for assignments, an assignment method was added to the auxiliary class that performs the real assignment on the embedded polymorphic object. There are no additional purposes of this auxiliary class other than wrapping this member.

Reconfigurable Object Version. A third version uses a reconfigurable object for computation. Since, reconfigurable objects support polymorphism and assignments, no wrapper class (like the shared object case) is required. Having these three implementation versions, multiple variants are built. They can be distinguished by:

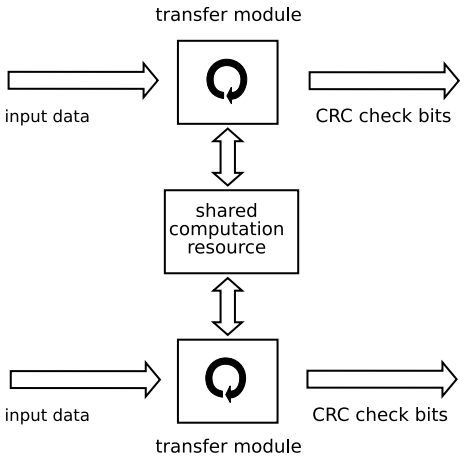

Figure 7.5: Block diagram: CRC benchmark with two processes

1. computations performed by one or multiple processes, and
2. computational resources being shared or not, and
3. computational resources being implemented statically or reconfigurable.

Table 7.3 gives an overview on the possible variants. Depending on the three distin-

User-def. processes	Exclusive or shared res.	Static or reconfig.	Object kind	Scheduler inside...
One	No difference	Static	`osss_polymorphic`	(None)
One	No difference	Reconf.	`osss_recon`	(None)
Multiple	Exclusive	Static	`osss_polymorphic`	(None)
Multiple	Exclusive	Reconf.	`osss_recon`	Reconf. control
Multiple	Shared	Static	`osss_shared`	Shared object
Multiple	Shared	Reconf.	`osss_recon`	Access control

Table 7.3: CRC benchmark: Implementation variants

guishing properties, different implementations are chosen. In case of one user-defined process using the computational resources, using shared objects is not feasible. Therefore, the static version is implemented directly using a polymorphic object, while the reconfigurable version uses a reconfigurable object. Also, it does not make sense to distinguish between use of shared and exclusive resources if there is only one user-defined process involved.

In the case of multiple user-defined processes and exclusive use of static resources, each process operates on its own polymorphic object. If the computational resource is to be implemented as a reconfigurable unit, the polymorphic object is replaced by a reconfigurable object. Since, there are multiple reconfigurable objects competing for the reconfiguration resources, this version contains a scheduler inside the reconfiguration controller.

If the computational resources are to be shared among processes, the implementation is modified again. In the case of a static implementation, a shared object is used. It contains a scheduler to put the access requests from the user-defined processes into a sequence. The reconfigurable variant uses a single reconfigurable object. This object also has a scheduler for the accesses but there is no scheduler inside the reconfiguration controller (since there is only one reconfigurable object).

Initial Version. Figure 7.6 gives a first impression of the overhead introduced by partial reconfiguration. The three columns show the area for the complete benchmark. The VHDL source files generated by *fossy* are processed by Xilinx XST tool. XST reports a slice count estimate for all files processed. The synthesis was done for the same Xilinx 4 device as used in the waveform generator benchmark earlier in this chapter. Note: These are not exact figures since the circuit is not placed and routed. Before placement and routing, the slice count is significantly over-estimated compared to the results after logic synthesis. The figures are rather meant to demonstrate trends and influences of changes on the circuit's area.

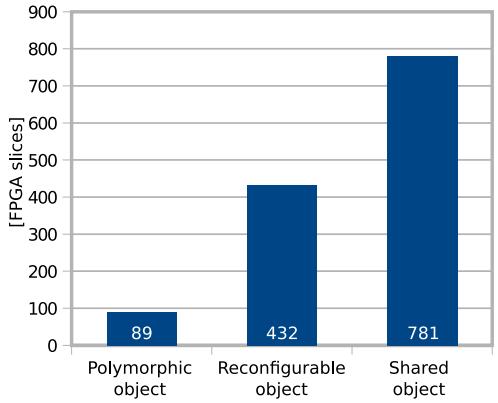

Figure 7.6: CRC: 3 implementations, slice count

All three bars refer to an implementation with only one user-defined process. The first bar shows the use of a polymorphic object. Since, a polymorphic object is no entity object, it does not contain any processes and its functionality can be inlined into the processes manipulating it. This allows much more gate-level optimization. The reason is that entity objects will be implemented with embedded processes communicating with the user-defined processes manipulating these entity objects. Synthesis tools usually implement individual control logic for these processes. Also, it is more difficult to optimize the datapath logic of these processes, since they are treated separately. As a consequence, the polymorphic object is superior in case of a single user-defined object.

The second bar shows the reconfigurable solution. It is considerably larger than the polymorphic object, since it is an entity object (and thus has its own processes inside). The third bar shows a shared object solution. It is somewhat artificial to implement the benchmark this way, since a shared object cannot show its advantages, if there is no sharing of resources. Nevertheless it is included to help putting things into perspective.

The reconfigurable object appears to be smaller than a shared object. This is misleading for three reasons. First, the reconfigurable object case includes the largest implementation of a reconfigurable module, but it does not include the board support package. The PDRC is missing in this case. In the Waveform generator example, the files from the board support package required 227 slices on the Xilinx Virtex 4 (after placement and routing). Second, an area equivalent for use of an external bitstream storage needs to be accounted for. In case of the board support package used before, this is the SRAM component. Third, the shared object implementation does not use any optimizations. Eike Grimpe[52] demonstrated how shared objects can be optimized. Its experiments were done with the predecessor tool of *fossy* . The current implementation of *fossy* does not include these optimizations.

Table 7.4 shows an analysis of the components used in the reconfigurable case. All user-defined code resides in a module called *transfer module*. *fossy* adds a module for an access controller and the platform independent reconfiguration controller (PIRC)

part. Additionally, a module implementing a reconfigurable area's logic is shown. Neither access controller nor PIRC include a scheduler. The access controller is connected to a single user-defined process and the PIRC is connected to a single access controller. A crossbar is omitted, since the only user-defined process is directly connected to the only reconfigurable area in this simple version of the CRC benchmark.

Module	Slices	Percentage
Transfer module	299	69.21%
Access controller	13	3.01%
PIRC	114	26.39%
Largest slot	6	1.39%

Table 7.4: CRC benchmark: Area breakdown of osss_recon version

The user-defined processes are modified by *fossy* . Object manipulation statements are replaced by a protocol for communication with the access controller and the slot. The state machine inside the user-defined processes is extended by states for the protocol steps.

Adding Further Processes. The first modification is to add further processes performing identical tasks. Table 7.5 shows the slice counts.

Implementation	1 process	2 processes	3 processes	4 processes
`osss_polymorphic`	89	154	238	307
`osss_recon`, exclusive	432	808	1192	-
`osss_recon`, shared	432	862	1220	-
`osss_shared`	781	1622	2429	-

Table 7.5: CRC benchmark: Increasing slice count with user-defined processes

The reconfigurable and shared object implementations were performed up to 3 processes only due to memory limitations of *fossy* . Figure 7.7 shows the table graphically. Obviously, the area demands for the reconfigurable versions do not increase as drastically, as the shared object version. And, the exclusive and shared cases of the reconfigurable version are comparable in size. This is worth investigating further.

Table 7.6 shows a breakdown of the reconfigurable case with one shared reconfigurable object being used by all user-defined processes. As one can see, the area increase is partially due to a crossbar being inserted from a two-process version on. This crossbar switches the communication between the slot (reconfigurable area) and the two user-defined processes. The crossbar grows in size with the number of user-defined processes. Since, it basically is a multiplexer structure, it would be worth investigating whether this could be optimized for a specific FPGA architecture. This crossbar will not grow in its size, if the user-defined processes make more intense use of the reconfigurable slot. It can be expected to be of less importance in larger designs. A second observation is that the access controller increases in size. It needs to be able to handle multiple user-defined processes, control the crossbar, and perform scheduling and permission handling. The task gets more complicated as the number of user-defined processes grows. The implementation of the reconfigurable area stays completely untouched. It is transparent for this module, how many user-defined processes potentially interact with the slot. Since, the number of access controllers is constant and no further reconfigurable areas are introduced with the user-defined processes, the PIRC stays constant in size.

Table 7.7 shows the counterpart with a growing number of user-defined processes each having their own reconfigurable object for exclusive use. Here, the area for the

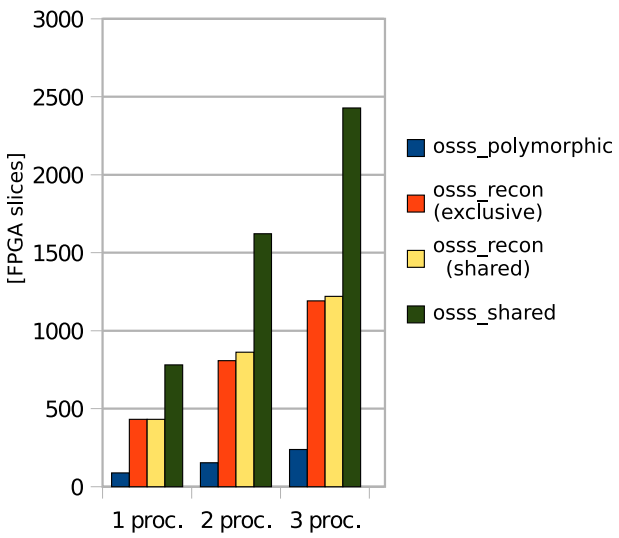

Figure 7.7: CRC: Adding processes

Breakdown `osss_recon`, shared	1 process	2 processes	3 processes
	[Amount of FPGA slices]		
Transfer module	299	527	792
Access controller	13	140	178
Crossbar	-	75	130
PIRC	114	114	114
Sum of largest slots	6	6	6
Summary	432	862	1220

Table 7.6: CRC Benchmark: Analysis of shared reconfigurable object version

Breakdown `osss_recon`, exclusive	1 process	2 processes	3 processes
	[Amount of FPGA slices]		
Transfer module(s)	299	598	897
Access controller(s)	13	26	39
Crossbar	-	-	-
PIRC	114	168	238
Sum of largest slots	6	12	18
Summary	432	804	1192

Table 7.7: CRC benchmark: Analysis of exclusive reconfigurable object version

access controllers grows almost linear. This can be expected, since each reconfigurable object receives a single accompanying access controller of the same complexity as the initial version. Since, there is no shared use of reconfigurable areas, no crossbars are inserted. The PIRC, however does grow in size. A scheduler is located inside the PIRC and it needs to handle multiple access controllers from the two-process version on. The transfer modules grow in size as well. This is mainly due to the increase in communication ports compared to the shared version.

Increase of CRC Classes. In this experiment, the number of CRC polynomials was increased. Each polynomial is implemented by its own class. Therefore, adding polynomials means adding classes to the inheritance tree. Table 7.8 shows the used polynomials. In the initial version, only CCITT-4 and CCITT-8 were used. Then, the benchmark was extended with the other entries in the table. The rightmost column shows the

Number	Name	Grade	Polynomial	FPGA slices
1	CCITT-4	5	$x^4 + x + 1$	4
2	CCITT-8	9	$x^8 + x^5 + x^4 + 1$	6
3	CCITT-16	17	$x^{16} + x^{12} + x^5 + 1$	7
4	XModem	17	$x^{16} + x^{15} + x^{10} + x^3$	4
5	IBM-CRC-16	17	$x^{16} + x^{15} + x^2 + 1$	4
6	CRC-24	25	$x^{24} + x^{23} + x^{18} + x^{17}$	12
	(IETF RFC2440)		$+x^{14} + x^{11} + x^{10} + x^7$	
			$+x^6 + x^5 + x^4 + x^3 + x + 1$	
7	CRC-32	33	$x^{32} + x^{26} + x^{23} + x^{22} + x^{16}$	22
			$+x^{12} + x^{11} + x^{10} + x^8 + x^7$	
			$+x^5 + x^4 + x^2 + x + 1$	

Table 7.8: CRC benchmark: Polynomials

number of slices required when implementing the CRC algorithm with that polynomial as a OSSS+R slot. Table 7.9 shows the area impact of adding further polynomials as numbers and Figure 7.8 gives a visual impression.

In case of a reconfigurable implementation, adding polynomials means that further variants of the reconfigurable area (one for each polynomial) are implemented. During OSSS+R synthesis, *fossy* generates more slot implementations, one for each class. When looking at the slice count for the reconfigurable solution, it is obvious that the transfer module requires the most area. The transfer module contains the user-defined clocked processes. The processes' state machine is modified by *fossy* . Further states are introduced for the protocol steps during communication with the access controller and the slot. It can be seen that the slice count for the transfer module is fluctuating by nearly 10%. This is likely due to the XST tool using heuristics for synthesis. It is reasonable that the slice count of the transfer module does not increase significantly. For the user-defined processes the only significant difference is that the datatype for identification of the polynomial class grows. Since, this number is encoded as a positive binary number (and not one-hot or similar), this grows very slowly. The growth appears to be smaller than the fluctuation due to XST synthesis. For the static solution using a polymorphic datatype, a different trend is visible. By adding a new class, the number of variants of the polymorphic datatype is increased. Since, the operations on polymorphic datatypes are implemented by inlining the functionality, the size of the transfer module grows steadily. Even when using a small algorithm like CRC, it can be seen how a static implementation grows in size with the increased functionality. A reconfigurable solution has an initial overhead but then grows slower in terms of area.

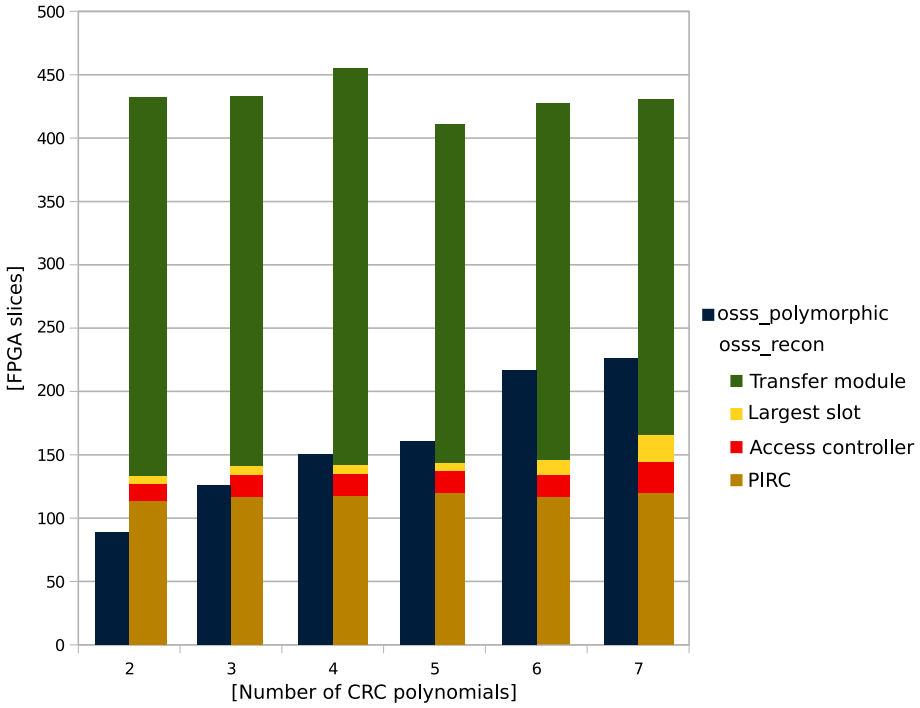

Figure 7.8: Area trend when adding further polynomials

Polynomials	2	3	4	5	6	7
PIRC	114	117	118	120	117	120
Access controller	13	17	17	17	17	24
Largest slot	6	7	7	7	12	22
Transfer module	299	292	313	267	281	265
Sum for `osss_recon`	432	433	455	411	427	431
`osss_polymorphic`	89	126	150	161	217	226

Table 7.9: CRC benchmark: Slice count when adding polynomials

7.4 Evaluation

The evaluation of OSSS+R follows the criteria introduced in Section 1.3. There are four criteria defined for the proposed approach:

1. Integration into existing tool-flows
2. Short development cycle
3. Design quality
4. Platform independence

These criteria are now discussed in more detail.

Criterion: Integration into Existing Tool-Flows

The chance for acceptance of any new methodology rises, if it fits into existing tool-flows. For the input side of the presented methodology, this clearly means conformance to common system description languages. As discussed in Section 2 this can be addressed by integration into a C/C++ and SystemC based flow. OSSS is a synthesizable subset of SystemC, enriched with extra elements to ease the transformation of C/C++ algorithmic descriptions into a system description. The proposed approach, called OSSS+R, is based on OSSS. Any SystemC model can be compiled and executed to simulate its behavior. Therefore, it is a requirement for OSSS+R to support simulation. Chapter 5 describes the simulation abilities of OSSS+R. It allows a cycle-accurate modeling and simulation of the final system.

After simulation, an abstract model needs to be refined to a more detailed, less abstract implementation model. This may require re-coding the model in a lower level language, e.g. Verilog or VHDL, for SystemC models. Alternatively, a synthesis tool may be used to aid this transformation. This part of the integration is addressed by the *fossy* tool. It was extended to transform a subset of OSSS+R into VHDL models at register-transfer level.

There are two models generated by *fossy* . First, a simulation model mimics the FPGA's behavior during reconfiguration. A multiplexer component and a generated reconfiguration controller (PDRC part) allow simulating the reconfiguration using standard VHDL simulators. This was shown using the Mentor Graphics ModelSim tool. This model is cycle accurate with the OSSS+R model.

The second model is intended for further processing by synthesis tools. It is a subset of the register-transfer level simulation model. It was demonstrated using Xilinx tools, that this model can be implemented on Xilinx Virtex 4 and 5 FPGAs.

A more in-depth discussion of the tool-flow can be found in Section 4.1. The *fossy* tool does not support all OSSS+R elements yet. However, it is possible to add the missing elements, but this was not done yet due to the implementation effort. In Chapter 6, a set of components is described that allows implementing all OSSS+R elements at a register-transfer level.

In summary, there is a way shown from C/C++ down to a register-transfer VHDL description. Its feasibility was demonstrated using the waveform generator benchmark. OSSS+R interfaces with a common algorithm specification language (C/C++) on its entry and with a common hardware description language (VHDL) as its output language.

Criterion: Short Development Cycle

To address the requirement for a short development time, OSSS+R was designed to permit a quick transformation of C/C++ models to OSSS+R. Here, it does not solve the general problems that arise when transforming a C/C++ model to SystemC. Instead, it offers library elements that support modeling reconfigurable elements. These elements are specified very similar to non-reconfigurable library elements.

As an example, the operators offered by the reconfigurable objects and permanent contexts match those of the polymorphic objects. Method invocations are possible by

using the operator −> in both cases and assignments from these objects have the same semantics. Both reconfigurable and polymorphic objects allow modeling of dynamic types. An implementation using polymorphic objects can be quickly transformed to use permanent contexts. The reverse transformation is also very quick.

Reconfigurable objects are shared, like shared objects in OSSS. When it comes to shared objects, the transformation from a static to a reconfigurable implementation is not as quick, since shared objects are not polymorphic and manipulated differently. On the other hand, as shown in the CRC benchmark, it is possible to embed polymorphic objects in shared objects to mimic the behavior of polymorphic shared objects.

Besides this support for manual design space exploration with little coding effort, the avoidance of potential pitfalls is important. This was addressed by hiding complexity inside the reconfigurable objects and permanent contexts. An intrinsic control structure ensures that certain types of mistakes cannot be made. For example, accessing a reconfigurable area without obtaining access rights is impossible using OSSS+R. This protection is even larger, if anonymous accesses are avoided. Using permanent contexts only, it is guaranteed that the correct dynamic state and logic configuration are put in place before the reconfigurable area performs its tasks. Access conflicts and reconfiguration conflicts are automatically resolved.

Bottlenecks in the reconfigurable systems can be found by using the OSSS+R logging facility and a graphical visualization tool. A set of optimization techniques allows improving run-time and area use.

To fully verify that the proposed methodology significantly shortens the development cycle, two designers would be needed to perform the same design task. One designer would be asked to use OSSS+R, the other person would use a traditional design flow. Having one person making both implementations in sequence would be a dramatic benefit for the second flow, whichever that may be, since the designer would have gained increased application knowledge by the first flow. Although it is difficult to perform a really fair test, it is reasonable to expect that disburdening the designer to specify partial reconfiguration in full detail would shorten design time. It means less specification effort and less potential for making mistakes.

When a dynamically partial reconfigurable system is to be designed, the following steps are to be performed. The steps marked with a minus symbol (-) are necessary whether OSSS+R is used or not, while a plus symbol (+) indicates aid by OSSS+R.

- \- Identify functionality to be used mutually exclusive.
- +/- Ensure loose coupling of this functionality.
- \+ Make static design parts aware of temporal unavailability of dynamic components.
- \+ Design and implementation of functionality to detect conditions for reconfigurations and trigger these.
- +/- Implementation of reconfiguration control logic.
- \+ In case of multiple reconfigurable regions: Arbitrate reconfiguration.
- \+ In case of shared use of reconfigurable regions: Arbitrate accesses and implement crossbars or busses for communication.
- \+ In case of stateful reconfigurable logic: Implement persistency.
- \+ Simulation: Implement two models, one for simulation and one for synthesis.
- \- Vendor specific steps (Bus-macro insertion, specially crafted top-level module, floorplanning, etc.).

The selection of functionality to be exchanged is to be done by the designer. One may extend the approach with a tool providing suggestions for this step, but this is not in the scope of this thesis.

Loose coupling of the functionality to be replaced is not done automatically in OSSS+R. It is achieved through requiring the functionality to be implemented as C++ classes. By doing so, the designer enables the *fossy* synthesis tool to generate components that are loosely coupled. The reconfigured functionality is not interwoven with

the processes using it, but implemented as a separate module by *fossy* .

Detecting conditions for reconfiguration is done by OSSS+R in two phases. First, user-defined processes reaching method invocation statements on reconfigurable objects, thereby state demand for a specific state of the reconfigurable object. For each method invocation, it is clear, what the preconditions are: What functionality needs to be configured to the FPGA area and what state does it need to be in? Second, the necessary preconditions are satisfied by reconfiguring the area (if necessary) and loading the correct state (if necessary). There is no need for the designer to specify this infrastructure.

The implementation of the reconfiguration control logic is a platform specific task. This is not done by the proposed approach. This logic is imported from the board support package. On the other hand, OSSS+R does however ease re-use of reconfiguration control logic over several applications by loosely coupling the board support package and the application logic.

If there is demand for multiple reconfigurable regions, there may be overlapping reconfiguration processes. Since, all FPGA devices currently on the market do not support concurrent reconfiguration, these processes need to be put into sequence. This is automatically done by inferring a scheduler. This scheduler may be customized by the designer, if necessary.

Reconfigurable regions are critical and valuable resources. This may lead to shared use of these regions by multiple user-defined processes. In such cases, a communication network between the user-defined processes and reconfigurable regions is to be implemented. This may be a point-to-point connection style and a set of multiplexers, or even a bus system. Additionally, there is a demand for resolving access conflicts. This is more complicated than resolving overlapping reconfigurations, since a faulty ordering of accesses may modify the application semantics, decrease performance or even cause deadlocks. Again, this step is automated by OSSS+R but may be influenced using locking mechanisms or custom schedulers on demand.

Reconfigured logic may be stateful or stateless. Stateless logic does not have any information to be preserved, if it is overwritten by different logic. A constant multiplier is an example for such a circuit. Examples for stateful algorithms are cryptography algorithms, Fourier transformations, checksum algorithms, and compression algorithms. Using these algorithms is possible, if they are not overwritten until their state is irrelevant. A MPEG decoder may be safely discarded after all data is decompressed. But whenever there is a demand to use the reconfigurable area earlier, the current state needs to be saved and restored afterwards. By providing persistent contexts, OSSS+R automates this task.

Using normal HDL design environments, models of reconfigurable circuits are usable either for simulation or for synthesis, but not for both purposes. The reason for this is that reconfiguration cannot be simulated using standard HDL simulators and therefore needs to be imitated by extra circuity. A logical approach would be to build a simulation model first, validate it and then modify it to be synthesizable. First, this modification is an extra design effort . Second, there is always a risk of semantically modifying the model in this step. OSSS+R provides high-level simulation abilities as in terms of a C++ library to be used in conjunction with the SystemC library. The very same model may then be synthesized to register transfer level. On this level, both a dedicated simulation version, as well as a synthesis version can be generated by *fossy* .

Finally, there are vendor-specific steps to be performed. For Xilinx, this is floorplanning of rectangular reconfigurable regions, introduction of bus-macros, a specially crafted top-level design instance and the creation of a constraint file.

This list demonstrates that not all, but many steps are supported or even automated by OSSS+R. Therefore, OSSS+R is shortening the design cycle compared to manual design.

Criterion: Design Quality

The third criterion is the quality of the resulting design. In the beginning of this chapter, the area use was taken to analyze the quality of OSSS+R designs. First, a waveform generator benchmark was used to show that a complete reconfigurable system can be implemented using a fraction of a today's typical FPGA. For the waveform generator benchmark, it was sufficient to use about 15% of the logic resources on a Xilinx Virtex 4 LX-25 device.

Using DPR introduces an area overhead. This is due to management logic, floorplanning considerations and the reconfiguration controller itself. The larger this overhead is, the more it compensates for the area saving due to hardware re-use. Although the acceptable overhead depends on the application to be implemented, it is likely that the overhead should not exceed the size of one implementation of the reconfigurable area. Cryptography algorithms are potential candidates to be implemented to be used mutually exclusive inside hardware accelerators. The survey[48] on the finalists competing to become the new *Advanced Encryption Standard* (AES) algorithm shows that their area roughly ranges from 1000 to 4500 FPGA slices on an Virtex I device. This device is a predecessor to the Virtex 4 and 5 devices used to evaluate OSSS+R. Although the slice usage numbers cannot be compared directly, the architectures are similar enough to allow comparing the orders of magnitude.

The only module in the waveform generator benchmark that contained user-defined processes and reconfigurable objects was the `Producer` module (184 slices). The platform independent part of the reconfiguration controller and board support package add an additional 336 slices. In total, 520 slices contain logic that *may* be inferred due to the use of partial reconfiguration. This is a very conservative and safe upper bound, since the producer module is doing some application logic as well. In addition, the waveform generator benchmark has two reconfigurable areas, which is not the simplest application, since it requires additional management overhead.

Reconfiguring algorithms with some 1000 slices of size (like the AES finalists) is unlikely to be infeasible, if the area overhead by the reconfiguration infrastructure is around one half of the smallest algorithm being exchanged.

A second benchmark, a cyclic redundancy check, was used to further analyse the area requirements of a OSSS+R design. It was shown how the individual components grow, when the design complexity is increased (more user-defined processes and more algorithms). From Figure 7.8 it can be seen that the management infrastructure grows moderately when adding further algorithms to be reconfigured. For the CRC benchmark, this growth was even smaller than the variation due to the synthesis heuristics by the XST tool. This shows that the proposed approach scales well when adding reconfigurable variants.

So far, there have been no investigations on how to fine-tune and optimize *fossy* synthesis with respect to FPGA targets. It is possible that the more regular structures (e.g. the multiplexers inside the crossbars) can be implemented in an optimized way. This would further reduce area usage.

Criterion: Platform Independence

The fourth criterion is the independence of a specific FPGA vendor and its tools. Right now, this has not been demonstrated, since the benchmark implementations relied on Xilinx devices and tools so far.

As discussed in Section 2.6.2, there are other FPGA architectures on the market, of which at least the Atmel AT94K devices appear to be suitable for OSSS+R. This device was used in the RECONF2 project to implement partial reconfiguration[85]. In an AT94K solution, the reconfiguration control is intended to be done by a microprocessor. In OSSS+R, the reconfiguration control is done by the access controller and reconfiguration controller components. A board support package for AT94K could

implement the platform dependent reconfiguration controller part (PDRC) on the microcontroller. This way, the software-PDRC would just need to receive reconfiguration requests from the PIRC and execute these. One would need to ensure, that the FPGA part and processor part would be able to perform a simple communication (but this can be expected on such a sophisticated component by Atmel). The application part itself is generated by *fossy* as platform independent code. There are no Xilinx specific components instantiated.

If dynamic partial reconfiguration is supported by more vendors in the future, it is likely that OSSS+R will be suitable to describe such systems.

Conclusion of the Evaluation

Section 7.4 analyzed the OSSS+R approach with respect to four goals.

The integration into the existing tools flows can be treated as a success. First, it fits into the C/C++ and SystemC modelling environment on the abstract level. Second, it provides a tool-assisted path down to a register-transfer level description, as required by common hardware design tools. This includes the Xilinx XST tool and the Xilinx EAPR flow used to demonstrate the synthesis.

The shortened development cycle was achieved by automating several steps in the design of the partial reconfigurable systems. Not only does this disburden the designer of these tasks, it makes it also possible to try out a reconfigurable solution. If partial reconfiguration shows to be not suitable for the application, it is quick to switch back to a static solution.

The answer to the question of an acceptable area overhead depends on the application to be implemented. It was shown using the waveform generator benchmark, that OSSS+R has an intrinsic area overhead that consumes a small fraction of current FPGA's resources. The CRC benchmark demonstrated that OSSS+R scales with the additional algorithms to be added to the set of reconfiguration variants. Therefore, it is likely that the OSSS+R infrastructure does not make DPR infeasible.

Finally, the OSSS+R synthesis does not rely on specific properties of Xilinx' EAPR flow or their FPGAs. Table 1.1 listed the required properties of potential FPGA devices that can be targets for OSSS+R.

7.5 Chapter Summary

This chapter briefly introduced the requirements of Xilinx' most current synthesis flow for dynamic partial reconfiguration. Then, an audio signal generator benchmark was used to demonstrate the feasibility of the proposed approach, starting from an initial C/C++ model to an operational implementation on a FPGA prototyping board. A second benchmark helped analyzing the impact of various design decisions on the resulting circuit size. Finally, the chapter was concluded with a discussion of the four goals of this work, and to which degree these were achieved.

Chapter 8

Conclusion

Summary

Dynamic partial reconfiguration is an adaption of the hardware to changing needs at runtime. It is a dynamic exchange of logic, while the system is in operation. This offers a new degree of flexibility for hardware design. It may enable maintenance upgrades in the field, improvements of robustness properties or, an increase in circuit area utilization.

Unfortunately, the design process of such systems is very cumbersome and tedious compared to traditional, static implementations. This is mainly rooted in the lack of expressiveness of traditional hardware description languages. Therefore, hardware design tools do not support dynamic partial reconfiguration. There are only a few experimental tool solutions, supporting partial reconfiguration to a certain degree.

This thesis presents an approach that eases the design of dynamic partial reconfigurable systems. An object oriented system description library (OSSS) is extended to allow modeling of such adaptive systems. The proposed set of modeling elements is called OSSS+R.

OSSS+R can be simulated to perform functional validation. The simulation library is based on SystemC and C/C++. It is shown that some elements of C++ models can be quickly transformed into a description of a reconfigurable system. This is done by preserving polymorphism on one hand and mutual exclusive use of objects on the other hand. Polymorphism allows describing adaptivity on a lower level of abstraction. Dynamic type changes in the C++ model are mapped to partial reconfigurations in the hardware model.

Mutual exclusive use of objects offers a more abstract description of reconfigurability. OSSS+R implements a virtual hardware concept for such objects by modeling them as permanent contexts. Permanent contexts show an unlimited lifetime and the behavior of independent objects. Nonetheless they are implemented mutually exclusive on the reconfigurable area by time-multiplexing. This concept is well-known from virtual memory and paging mechanisms in personal computers.

The proposed approach includes aids to optimize a model for runtime and area use. This includes user-defined and history-aware scheduling algorithms and state-dependent locking mechanisms. These features help tune the scheduling of reconfigurations and accesses in an OSSS+R model to the application's needs. Support for multiple reconfigurable areas and shared use of these offers further flexibility. Finally, state space reduction techniques help limiting the area used by the reconfigurable system.

OSSS+R can be synthesized to a register-transfer level model. To demonstrate that this can be done automatically, parts of the transformation are implemented in a tool called *fossy* . It accepts OSSS+R models and generates register-transfer level VHDL

models. The generated models are cycle accurate with the original OSSS+R model.

There are four distinguishing features of this work:

Firstly, an integration into existing tool flows. The entry of the proposed flow is a C/C++ and SystemC model. C and C++ are well-established software programming languages, which are heavily used for algorithmic specification. SystemC is a system modeling library, which is increasingly popular.

Secondly, avoidance of re-coding at different levels of abstraction allows quick modeling. The proposed approach offers library elements, which ease the transformation of a system specification's hardware parts into a hardware description using reconfiguration. The resulting description, OSSS+R, may then be automatically transformed into register-transfer level VHDL code by the *fossy* tool. Simulation abilities (before and after synthesis to VHDL) allow model validation.

Thirdly, it was shown that the use of the proposed approach only causes reasonable hardware overhead. A reconfigurable audio generator system was implemented on an FPGA board. It used less than 15% of hardware resources provided by a Xilinx Virtex 4 LX-25 FPGA.

The fourth feature is the platform independence of the proposed approach. Although the feasibility was only shown using Xilinx devices, there is no dependency on devices of this vendor. The approach rather relies on a limited set of properties like deterministic reconfiguration times or partial configurability in general.

The feasibility of the proposed approach was demonstrated by implementing a design as a C++ model and then performing all proposed steps. Finally, the model was implemented on a FPGA prototyping platform.

Outlook

The scientific community is very actively looking for applications that benefit from dynamic partial reconfiguration. It has been discussed in this work, that suitable applications must have a certain set of properties in order to show a substantial benefit. Furthermore, the effort to be spent in the design process itself must be comparable to those for traditional non-reconfigurable systems. Product development cycles and product lifetimes of new systems are ever decreasing. Therefore, rapid development is gaining importance as well.

In this thesis an approach was presented that shows a way how to reduce development times for reconfigurable systems. There are many extensions possible that would further increase the benefit that stems from such an approach. For example, making communication links between the different components of an OSSS+R model customizable would increase its flexibility. Integrating synthesizable channels and transactors would allow introducing bus connections instead of fixed point-to-point connections. This would allow the designer to trade off the small communication latency of a point-to-point protocol against the flexibility of busses. As a second improvement, a system with fixed schedules and guaranteed object switch times could make OSSS+R suitable for hard real-time systems. It is likely that most changes required by this change could be done without changing the OSSS+R syntax. Thirdly, more control over the reconfiguration data source could lead to support for systems allowing in-the-field maintenance. To install an upgrade, the configuration data source could be blocked, updated and released. A set of rules for permitted changes in the classes' state spaces would be required to avoid breaking the application while it is running. One example could be to allow only adding transient attributes.

All of these additions are beyond the scope of this thesis.

Appendix A

Simulation Timing Testcase

This appendix contains an OSSS+R testcase that checks the simulation timing against the specification.

```
1  #include <osss.h>
2  #include <iostream>
3  #include <typeinfo>
4
5  using std::string;
6  using std::cout;
7  using std::endl;
8  using std::flush;
9
10 // Some definitions to implement CTHREADs
11 // with THREADs, if requested.
12 #if defined( FORCE_THREAD_PROCESSES )
13 #undef SC_CTHREAD
14 #define SC_CTHREAD( proc, clock ) \
15   SC_THREAD( proc ); \
16   sensitive << clock.pos()
17 #define reset_signal_is( signal, level )
18 #endif
19
20 // For better readability, use symbolic name
21 // for clock
22 #define CLOCK_PERIOD sc_time(1, SC_NS)
23
24 // Reconfiguation and copy times for classes A, B and C
25 #define CLASS_A_RECONF_TIME (CLOCK_PERIOD * 100)
26 #define CLASS_B_RECONF_TIME (CLOCK_PERIOD * 200)
27 #define CLASS_C_RECONF_TIME (CLOCK_PERIOD * 300)
28 #define CLASS_A_COPY_TIME   (CLOCK_PERIOD *  10)
29 #define CLASS_B_COPY_TIME   (CLOCK_PERIOD *  20)
30 #define CLASS_C_COPY_TIME   (CLOCK_PERIOD *  30)
31
32 // Now define constants for the protocol timing
33
34 #define DO_DONE_OVERHEAD     (CLOCK_PERIOD * 2)
35
36 // No recon scheduler overhead: There is no scheduler
37 // since we have only one user process as client
38 //#define RECON_SCHEDULER_OVERHEAD   CLOCK_PERIOD
39 #define RECON_SCHEDULER_OVERHEAD   SC_ZERO_TIME
40
41 #define DEVICE_SCHEDULER_OVERHEAD  CLOCK_PERIOD
42 #define RECON_COLLECT_CYCLE        CLOCK_PERIOD
43 #define DEVICE_COLLECT_CYCLE       CLOCK_PERIOD
44 #define SINGLE_OP_OVERHEAD         CLOCK_PERIOD
45 #define SLOT_RESET_CYCLE           CLOCK_PERIOD
46 #define PDRC_DELAY_CYCLE           CLOCK_PERIOD
47
```

```
48 #define RETURN_PERM_OVERHEAD          SC_ZERO_TIME
49
50 #define PERMISSION_PROTOCOL_OVERHEAD\
51 /* request perm */      (    /* up <-> ac */\
52                              DO_DONE_OVERHEAD\
53                            + RECON_SCHEDULER_OVERHEAD\
54                            + RECON_COLLECT_CYCLE\
55                              /* multi->single: new grant! */\
56                            + SINGLE_OP_OVERHEAD\
57 /* Return permission */  + RETURN_PERM_OVERHEAD)
58
59 #define COPY_PROTOCOL_OVERHEAD (  /* save: ac <-> storage */\
60                                   DO_DONE_OVERHEAD\
61                                   /* save: "accept == false" cycle ,\
62                                        slot <-> storage */\
63                                 + DO_DONE_OVERHEAD)
64
65 /* Add configuration time for class separately! */
66 #define RECONF_PROTOCOL_OVERHEAD ( /* ac <-> pirc */\
67                                    DO_DONE_OVERHEAD\
68                                    /* pirc <-> pdrc */\
69                                  + DO_DONE_OVERHEAD\
70                                  + DEVICE_COLLECT_CYCLE\
71                                  + DEVICE_SCHEDULER_OVERHEAD\
72                                  + SLOT_RESET_CYCLE\
73                                  + PDRC_DELAY_CYCLE )
74
75 #define METHOD_INVOCATION_OVERHEAD DO_DONE_OVERHEAD
76
77
78 // Variable to remember start times
79 sc_time last_measurement;
80
81
82 // Check a process duration against the expectations
83 bool expect(string msg, sc_time expected_duration)
84 {
85   cout << sc_time_stamp() << " " << msg << ": " << flush;
86   sc_time measured_duration  = sc_time_stamp() - last_measurement;
87   last_measurement = sc_time_stamp();
88   if (measured_duration == expected_duration)
89   {
90     cout << "[" << measured_duration << "]: PASSED " << endl;
91     // Test successful.
92     return true;
93   }
94   // else:
95   cout << "measured [" << measured_duration
96        << "] != expected [" << expected_duration
97        << "], ";
98   if ( expected_duration > measured_duration )
99   {
100    cout << (expected_duration - measured_duration)
101         << " too quick: FAILED!" << endl;
102  }
103  else
104  {
105    cout << (measured_duration - expected_duration)
106         << " too slow: FAILED!" << endl;
107  }
108  // Test failed.
109  return false;
110 }
111
112
113
114
115
```

```
116 // All tests are started at the beginning of
117 // a microsecond. This method waits until the
118 // next microsecond of simulation time starts.
119 void await()
120 {
121   int i = (int)(sc_time_stamp() / sc_time(1, SC_US));
122   sc_time t = sc_time(1+i, SC_US);
123   cout << endl;
124   while (t > sc_time_stamp())
125   {
126     sc_core::wait();
127   }
128   // Start measurement.
129   last_measurement = sc_time_stamp();
130 }
131
132
133 // An auxillary base class which provides
134 // a method to record its invocation time,
135 // elapse a specified duration and record
136 // its finishing time.
137 class UserBaseClass : public osss_object
138 {
139   public:
140
141   UserBaseClass()
142   {
143     OSSS_BASE_CLASS( osss_object );
144   }
145   virtual ~UserBaseClass()
146   {
147   }
148
149   virtual
150   void
151   test(sc_time &     start_time,
152        unsigned int wait_cycle_count,
153        sc_time &     end_time)
154   {
155     start_time = sc_time_stamp();
156     if ( wait_cycle_count > 0 )
157     {
158       wait( wait_cycle_count );
159     }
160     end_time = sc_time_stamp();
161   }
162 };
163
164
165 // A helper macro to create custom classes
166 #define CREATE_DERIVED_CLASS( classname )\
167 class classname : public UserBaseClass\
168 {\
169   public:\
170   classname() : UserBaseClass()\
171   {\
172     OSSS_BASE_CLASS( UserBaseClass );\
173   }\
174   virtual ~classname()\
175   {}\
176 }
177
178 // Create three custom classes
179 CREATE_DERIVED_CLASS( A );
180 CREATE_DERIVED_CLASS( B );
181 CREATE_DERIVED_CLASS( C );
182
183
```

```
184 SC_MODULE( my_module )
185 {
186   public:
187   // Two reconfigurable objects and four
188   // contexts to implement the tests.
189   osss_recon< UserBaseClass > recon_one;
190   osss_recon< UserBaseClass > recon_two;
191   osss_context< A > cA1;
192   osss_context< A > cA2;
193   osss_context< B > cB;
194   osss_context< C > cC;
195
196   sc_in< bool > clock;
197   sc_in< bool > reset;
198
199   void testproc()
200   {
201     expect("Test␣procedure␣start", SC_ZERO_TIME);
202
203     recon_one = A();
204     expect("Assignment␣to␣temporary␣context␣[sCr]",
205               PERMISSION_PROTOCOL_OVERHEAD
206             + RECONF_PROTOCOL_OVERHEAD
207             + CLASS_A_RECONF_TIME
208             + METHOD_INVOCATION_OVERHEAD /* Execute assignment */
209             );
210
211     await();
212     cA1 = A();
213     expect("Assignment␣to␣disabled␣context␣[scr]",
214               PERMISSION_PROTOCOL_OVERHEAD
215             + METHOD_INVOCATION_OVERHEAD /* Execute assignment */
216             );
217
218     await();
219     cA1 = A();
220     expect("Assignment␣to␣enabled␣context␣[scr]",
221               PERMISSION_PROTOCOL_OVERHEAD
222             + METHOD_INVOCATION_OVERHEAD /* Execute assignment */
223             );
224
225
226     await();
227     sc_time start, stop;
228     sc_time now = sc_time_stamp();
229     cA1->test(start, 42, stop);
230     expect("Method␣invocation␣on␣enabled␣context␣[scr]",
231               PERMISSION_PROTOCOL_OVERHEAD
232             + METHOD_INVOCATION_OVERHEAD /* Execute call */
233             + (CLOCK_PERIOD * 42)
234             );
235     sc_time expected_cycles =
236                   PERMISSION_PROTOCOL_OVERHEAD
237                 /* returning perm. is performed later: */
238                 - SINGLE_OP_OVERHEAD
239                 + METHOD_INVOCATION_OVERHEAD
240                 /* signalling "method performed" is idicated later: */
241                 - SINGLE_OP_OVERHEAD;
242     if ((start - now) == expected_cycles)
243     {
244       OSSS_MESSAGE(true, "Execution␣start␣time␣ok␣["
245                          << (start-now)
246                          << "]:␣PASSED");
247     }
248     else
249     {
250       OSSS_MESSAGE(true, "Execution␣start␣time␣wrong␣[expected␣"
251                          << expected_cycles
```

```
252                             << ",␣measured␣"
253                             << (start-now)
254                             << "]:␣FAILED");
255       }
256       if ((stop - start) == (CLOCK_PERIOD * 42))
257       {
258         OSSS_MESSAGE(true, "Execution␣duration␣ok␣["
259                         << (stop - start)
260                         << "]:␣PASSED");
261       }
262       else
263       {
264         OSSS_MESSAGE(true, "Execution␣duration␣wrong␣[expected␣"
265                             << (CLOCK_PERIOD * 42)
266                             << ",␣measured␣"
267                             << (stop - start)
268                             << "]:␣FAILED");
269       }
270
271       // Switch to cA2
272       await();
273       cA2 = A();
274       expect("Assignment␣to␣context␣[Scr]",
275                 PERMISSION_PROTOCOL_OVERHEAD
276               + COPY_PROTOCOL_OVERHEAD // save cA1
277               + CLASS_A_COPY_TIME
278               + METHOD_INVOCATION_OVERHEAD /* Execute assignment */
279               );
280
281       // Next test: assign something to a disabled context
282       await();
283       cB = B();
284       expect("Assignment␣to␣context␣[SCr]",
285                 PERMISSION_PROTOCOL_OVERHEAD
286               + COPY_PROTOCOL_OVERHEAD // save cA2
287               + RECONF_PROTOCOL_OVERHEAD
288               + CLASS_A_COPY_TIME
289               + CLASS_B_RECONF_TIME
290               + METHOD_INVOCATION_OVERHEAD /* Execute assignment */
291               );
292
293       // Calling methods on disabled context
294       await();
295       cA1->test(start, 42, stop);
296       expect("Method␣invocation␣on␣disabled␣context␣[scr]",
297                 PERMISSION_PROTOCOL_OVERHEAD
298               + CLASS_B_COPY_TIME
299               + COPY_PROTOCOL_OVERHEAD // save
300               + RECONF_PROTOCOL_OVERHEAD
301               + CLASS_A_RECONF_TIME
302               + COPY_PROTOCOL_OVERHEAD // restore
303               + CLASS_A_COPY_TIME
304               + METHOD_INVOCATION_OVERHEAD /* Execute call */
305               + (CLOCK_PERIOD * 42)
306               );
307
308       // Assign from one enabled context to another disabled one
309       await();
310       cA2 = cA1;
311       expect("Assignment␣of␣enabled␣context␣to␣a␣disabled␣one",
312                 2*PERMISSION_PROTOCOL_OVERHEAD
313               + COPY_PROTOCOL_OVERHEAD // save cA1
314               + CLASS_A_COPY_TIME
315               + 2*METHOD_INVOCATION_OVERHEAD /* Assignment call */
316               );
317
318       sc_stop();
319     }
```

```
320   SC_CTOR( my_module )
321   {
322     SC_CTHREAD(testproc, clock);
323     reset_signal_is(reset, true);
324     uses(recon_one); uses(recon_two);
325
326     recon_one.clock_port(clock);
327     recon_two.clock_port(clock);
328     recon_one.reset_port(reset);
329     recon_two.reset_port(reset);
330
331     cA1.bind(recon_one);
332     cA2.bind(recon_one);
333     cB.bind(recon_one);
334     cC.bind(recon_one);
335
336     last_measurement = SC_ZERO_TIME;
337   }
338 };
339
340 int sc_main(int /*argc*/, char * /*argv*/[])
341 {
342   sc_clock clock("main_clock", CLOCK_PERIOD);
343   sc_signal< bool > reset;
344
345   osss::osss_device_type my_device_type("My FPGA device type");
346   osss::osss_device my_device(my_device_type, "my_device");
347   OSSS_DECLARE_TIME(my_device_type, A, CLASS_A_COPY_TIME, CLASS_A_RECONF_TIME);
348   OSSS_DECLARE_TIME(my_device_type, B, CLASS_B_COPY_TIME, CLASS_B_RECONF_TIME);
349   OSSS_DECLARE_TIME(my_device_type, C, CLASS_C_COPY_TIME, CLASS_C_RECONF_TIME);
350   my_device.clock_port(clock);
351   my_device.reset_port(reset);
352
353   my_module mm("my_module");
354   mm.clock(clock);
355   mm.reset(reset);
356   mm.recon_one( my_device );
357   mm.recon_two( my_device );
358
359   sc_start();
360   return EXIT_SUCCESS;
361 }
```

The testcase performs various operations on the anonymous access object and on persistent contexts. Each time it checks the duration against the specified timing. The generated output is shown in the following listing:

```
1                SystemC 2.2.0 --- Jan 10 2008 16:04:22
2          Copyright (c) 1996-2006 by all Contributors
3                       ALL RIGHTS RESERVED
4 0 s Test procedure start: [0 s]: PASSED
5 114 ns Assignment to temporary context [sCr]: [114 ns]: PASSED
6
7 1006 ns Assignment to disabled context [scr]: [6 ns]: PASSED
8
9 2006 ns Assignment to enabled context [scr]: [6 ns]: PASSED
10
11 3048 ns Method invocation on enabled context [scr]: [48 ns]: PASSED
12 3048 ns~12193 my_module.testproc @ testproc(): Execution start time ok [4 ns]: PASSED
13 3048 ns~12193 my_module.testproc @ testproc(): Execution duration ok [42 ns]: PASSED
14
15 4020 ns Assignment to context [Scr]: [20 ns]: PASSED
16
17 5228 ns Assignment to context [SCr]: [228 ns]: PASSED
18
19 6194 ns Method invocation on disabled context [scr]: [194 ns]: PASSED
20
21 7026 ns Assignment of enabled context to a disabled one: [26 ns]: PASSED
22 SystemC: simulation stopped by user.
```

Appendix B

RTL Simulation Testcase

This appendix presents one of the testcases used to verify the correctness of the simulation model generated by *fossy* . The following code is first compiled using the OSSS+R simulation library and executed to obtain a reference output. Then, the same input is translated into a RTL simulation model using *fossy* . The generated output is compiled and executed, too. Finally, its output is checked against the OSSS+R reference output.

The testcase presented here verifies that virtual method calls are correctly translated. The first listing shows a configuration file containing auxillary macros and definitions to improve the readability of the following code.

```
1  #ifndef DISPATCH_CONFIG_H
2  #define DISPATCH_CONFIG_H
3
4  // This datatype is used to identify the
5  // classes and methods during runtime.
6  #define CODEC_ID_TYPE sc_bigint<32>
7
8  /* Numeric encoding of method properties:
9     V = virtual
10    N = non-virtual
11    P = pure virtual
12    M = missing */
13
14 #define V_MARK 1
15 #define N_MARK 2
16 #define P_MARK 4
17 #define M_MARK 8
18
19 // Class identification numbers:
20 #define A_MARK 1
21 #define B_MARK 2
22 #define C_MARK 4
23 #define NO_MARK 0
24
25 #define ERROR_MARK 0xFFFF
26
27 // Ausillary methods to pack and
28 // unpack indentification data.
29 inline int
30 code(int classMark,
31      int mAMark,
32      int mBMark,
33      int mCMark)
34 {
35   return classMark
36        + (mAMark << 4)
37        + (mBMark << 8)
38        + (mCMark << 12);
39 }
```

```
40
41 inline int
42 classMark(int coded)
43 {
44   return coded & 0xF;
45 }
46
47 inline
48 int
49 aMark(int coded)
50 {
51   return (coded >> 4) & 0xF;
52 }
53
54 inline
55 int
56 bMark(int coded)
57 {
58   return (coded >> 8) & 0xF;
59 }
60
61 inline
62 int
63 cMark(int coded)
64 {
65   return (coded >> 12) & 0xF;
66 }
67
68 #include <osss>
69 using namespace osss;
70
71 // The following lines define three classes, A, B, and C.
72 class A : public osss_object
73 {
74   private:
75   // This is necessary so far, since fossy has problems
76   // with classes of zero bit state size.
77   bool work_around_a_bug;
78
79   public:
80   A(){ OSSS_BASE_CLASS( osss_object ); }
81
82   virtual int m_vvv(){ return code(A_MARK, V_MARK, V_MARK, V_MARK); }
83   virtual int m_vvn(){ return code(A_MARK, V_MARK, V_MARK, N_MARK); }
84   virtual int m_vvm(){ return code(A_MARK, V_MARK, V_MARK, M_MARK); }
85   virtual int m_vnv(){ return code(A_MARK, V_MARK, N_MARK, V_MARK); }
86   virtual int m_vnn(){ return code(A_MARK, V_MARK, N_MARK, N_MARK); }
87   virtual int m_vnm(){ return code(A_MARK, V_MARK, N_MARK, M_MARK); }
88   virtual int m_vmv(){ return code(A_MARK, V_MARK, M_MARK, V_MARK); }
89   virtual int m_vmn(){ return code(A_MARK, V_MARK, M_MARK, N_MARK); }
90   virtual int m_vmm(){ return code(A_MARK, V_MARK, M_MARK, M_MARK); }
91
92           int m_nvv(){ return code(A_MARK, N_MARK, V_MARK, V_MARK); }
93           int m_nvn(){ return code(A_MARK, N_MARK, V_MARK, N_MARK); }
94           int m_nvm(){ return code(A_MARK, N_MARK, V_MARK, M_MARK); }
95           int m_nnv(){ return code(A_MARK, N_MARK, N_MARK, V_MARK); }
96           int m_nnn(){ return code(A_MARK, N_MARK, N_MARK, N_MARK); }
97           int m_nnm(){ return code(A_MARK, N_MARK, N_MARK, M_MARK); }
98           int m_nmv(){ return code(A_MARK, N_MARK, M_MARK, V_MARK); }
99           int m_nmn(){ return code(A_MARK, N_MARK, M_MARK, N_MARK); }
100          int m_nmm(){ return code(A_MARK, N_MARK, M_MARK, M_MARK); }
101 };
102
103
104
105
106
107
```

```
108 class B : public A
109 {
110   public:
111   B(){ OSSS_BASE_CLASS( A ); }
112
113   virtual int m_vvv(){ return code(B_MARK, V_MARK, V_MARK, V_MARK); }
114   virtual int m_vvn(){ return code(B_MARK, V_MARK, V_MARK, N_MARK); }
115   virtual int m_vvm(){ return code(B_MARK, V_MARK, V_MARK, M_MARK); }
116           int m_vnv(){ return code(B_MARK, V_MARK, N_MARK, V_MARK); }
117           int m_vnn(){ return code(B_MARK, V_MARK, N_MARK, N_MARK); }
118           int m_vnm(){ return code(B_MARK, V_MARK, N_MARK, M_MARK); }
119
120   virtual int m_nvv(){ return code(B_MARK, N_MARK, V_MARK, V_MARK); }
121   virtual int m_nvn(){ return code(B_MARK, N_MARK, V_MARK, N_MARK); }
122   virtual int m_nvm(){ return code(B_MARK, N_MARK, V_MARK, M_MARK); }
123           int m_nnv(){ return code(B_MARK, N_MARK, N_MARK, V_MARK); }
124           int m_nnn(){ return code(B_MARK, N_MARK, N_MARK, N_MARK); }
125           int m_nnm(){ return code(B_MARK, N_MARK, N_MARK, M_MARK); }
126
127   virtual int m_mvv(){ return code(B_MARK, M_MARK, V_MARK, V_MARK); }
128   virtual int m_mvn(){ return code(B_MARK, M_MARK, V_MARK, N_MARK); }
129   virtual int m_mvm(){ return code(B_MARK, M_MARK, V_MARK, M_MARK); }
130           int m_mnv(){ return code(B_MARK, M_MARK, N_MARK, V_MARK); }
131           int m_mnn(){ return code(B_MARK, M_MARK, N_MARK, N_MARK); }
132           int m_mnm(){ return code(B_MARK, M_MARK, N_MARK, M_MARK); }
133 };
134
135 class C : public B
136 {
137   public:
138   C(){ OSSS_BASE_CLASS( B ); }
139
140   virtual int m_vvv(){ return code(C_MARK, V_MARK, V_MARK, V_MARK); }
141           int m_vvn(){ return code(C_MARK, V_MARK, V_MARK, N_MARK); }
142   virtual int m_vnv(){ return code(C_MARK, V_MARK, N_MARK, V_MARK); }
143           int m_vnn(){ return code(C_MARK, V_MARK, N_MARK, N_MARK); }
144   virtual int m_vmv(){ return code(C_MARK, V_MARK, M_MARK, V_MARK); }
145           int m_vmn(){ return code(C_MARK, V_MARK, M_MARK, N_MARK); }
146
147   virtual int m_nvv(){ return code(C_MARK, N_MARK, V_MARK, V_MARK); }
148           int m_nvn(){ return code(C_MARK, N_MARK, V_MARK, N_MARK); }
149   virtual int m_nnv(){ return code(C_MARK, N_MARK, N_MARK, V_MARK); }
150           int m_nnn(){ return code(C_MARK, N_MARK, N_MARK, N_MARK); }
151   virtual int m_nmv(){ return code(C_MARK, N_MARK, M_MARK, V_MARK); }
152           int m_nmn(){ return code(C_MARK, N_MARK, M_MARK, N_MARK); }
153
154   virtual int m_mvv(){ return code(C_MARK, M_MARK, V_MARK, V_MARK); }
155           int m_mvn(){ return code(C_MARK, M_MARK, V_MARK, N_MARK); }
156   virtual int m_mnv(){ return code(C_MARK, M_MARK, N_MARK, V_MARK); }
157           int m_mnn(){ return code(C_MARK, M_MARK, N_MARK, N_MARK); }
158 };
159
160 #ifndef EXIT_SUCCESS
161 #define EXIT_SUCCESS 0
162 #endif
163
164 #endif // DISPATCH_CONFIG_H
```

The configuration listing shown above defined three classes, `A`, `B`, and `C`. They inherit methods from each other where `A` is the parent class. The methods are named `m_xyz` where `x`, `y`, and `z` indicate the virtuality in the classes `A`, `B`, and `C`. According to the table at the beginning of this file, `v` is a virtual method, `n` is non-virtual etc. Each method returns it properties as a packed numeric value. This clearly identifies the class and method being invoked. If everything works fine, the method calls in OSSS+R and after fossy synthesis do match.

The following listing shows the module to perform the test. It periodically reads the `pi_method_id` port which tells the design under test which method to invoke. The

method invocation returns an encoded number indicating which method was executed. That number is returned using the `po_method_id` port. The testbench then checks whether the correct method was executed.

```
#include "systemc.h"

SC_MODULE(dispatchModule)
{
  // Encoded method IDs from and to testbench
  sc_in< CODEC_ID_TYPE >     pi_method_id;
  sc_out< CODEC_ID_TYPE >    po_method_id;
  // Ports for handshaking
  sc_in< bool >    pi_req;
  sc_out< bool >   po_ack;

  osss::osss_recon< B > ro;

  void xProc()
  {
    po_ack.write(false);
    wait(); // await end of reset

    // Install a most derived object of type C
    ro = C();
    while (true)
    {
      if (pi_req.read() == true)
      {
        int requested_id = pi_method_id.read().to_int();
        int a = aMark(requested_id);
        int b = bMark(requested_id);
        int c = cMark(requested_id);
        int calculated_id = ERROR_MARK;
        switch (a)
        { case V_MARK :
            switch (b)
            { case V_MARK :
                switch (c)
                { case V_MARK : calculated_id = ro->m_vvv(); break;
                  case N_MARK : calculated_id = ro->m_vvn(); break;
                  case M_MARK : calculated_id = ro->m_vvm(); break;
                }
                break;
              case N_MARK :
                switch (c)
                { case V_MARK : calculated_id = ro->m_vnv(); break;
                  case N_MARK : calculated_id = ro->m_vnn(); break;
                  case M_MARK : calculated_id = ro->m_vnm(); break;
                }
                break;
              case M_MARK :
                switch (c)
                { case V_MARK : calculated_id = ro->m_vmv(); break;
                  case N_MARK : calculated_id = ro->m_vmn(); break;
                  case M_MARK : calculated_id = ro->m_vmm(); break;
                }
                break;
            }
            break;
          case N_MARK :
            switch (b)
            { case V_MARK :
                switch (c)
                { case V_MARK : calculated_id = ro->m_nvv(); break;
                  case N_MARK : calculated_id = ro->m_nvn(); break;
                  case M_MARK : calculated_id = ro->m_nvm(); break;
                }
                break;
```

```
65                case N_MARK :
66                  switch (c)
67                  { case V_MARK : calculated_id = ro->m_nnv(); break;
68                    case N_MARK : calculated_id = ro->m_nnn(); break;
69                    case M_MARK : calculated_id = ro->m_nnm(); break;
70                  }
71                  break;
72                case M_MARK :
73                  switch (c)
74                  { case V_MARK : calculated_id = ro->m_nmv(); break;
75                    case N_MARK : calculated_id = ro->m_nmn(); break;
76                    case M_MARK : calculated_id = ro->m_nmm(); break;
77                  }
78                  break;
79              }
80              break;
81            /* Pure virtual methods cannot be invoked.
82            case P_MARK :
83              switch (b)
84              { case V_MARK :
85                  switch (c)
86                  { case V_MARK : calculated_id = ro->m_pvv(); break;
87                    case N_MARK : calculated_id = ro->m_pvn(); break;
88                    case M_MARK : calculated_id = ro->m_pvm(); break;
89                  }
90                  break;
91                case N_MARK :
92                  switch (c)
93                  { case V_MARK : calculated_id = ro->m_pnv(); break;
94                    case N_MARK : calculated_id = ro->m_pnn(); break;
95                    case M_MARK : calculated_id = ro->m_pnm(); break;
96                  }
97                  break;
98              }
99              break;
100           */
101           case M_MARK :
102             switch (b)
103             { case V_MARK :
104                 switch (c)
105                 { case V_MARK : calculated_id = ro->m_mvv(); break;
106                   case N_MARK : calculated_id = ro->m_mvn(); break;
107                   case M_MARK : calculated_id = ro->m_mvm(); break;
108                 }
109                 break;
110               case N_MARK :
111                 switch (c)
112                 { case V_MARK : calculated_id = ro->m_mnv(); break;
113                   case N_MARK : calculated_id = ro->m_mnn(); break;
114                   case M_MARK : calculated_id = ro->m_mnm(); break;
115                 }
116                 break;
117             }
118             break;
119         } // switch a
120         po_method_id.write( calculated_id );
121         po_ack.write( true );
122         wait();
123         while (pi_req.read() == true){ wait(); }
124         po_ack.write( false );
125       }
126       wait();
127     } // while
128   }
129
130   sc_in< bool > clock;
131   sc_in< bool > reset;
132
```

```
133   SC_CTOR(dispatchModule) : ro("ro"),
134                             pi_method_id("pi_method_id"),
135                             pi_req("pi_req"),
136                             po_method_id("po_method_id"),
137                             po_ack("po_ack")
138   {
139     ro.clock_port(clock);
140     ro.reset_port(reset);
141
142     SC_CTHREAD(xProc, clock.pos());
143     reset_signal_is(reset, true);
144     uses(ro, 42);
145   }
146 };
```

The third listing shows the testbench. It is implemented in a way that it passes through fossy, allowing the tool to analyse the `sc_main` method. Binding statements and device type definitions may also be placed there, as this testbench demonstrates. Fossy defines the `SC_SYNTHESIS` symbol, allowing the preprocessor to hide some statements.

```
1  #define OSSS_BLUE
2
3  #ifndef MODELSIM
4  #include <osss>
5  using namespace osss;
6  #else
7  #define OSSS_MESSAGE(a,b)
8  #endif
9
10 #include "dispatch_config.h"
11
12 #ifndef SC_SYNTHESIS
13 std::ofstream logfile;
14 #endif
15
16 #define VERBOSE_DEBUG_TB_MODE true
17
18 // Switch to include the synthesised design
19 // under test (fossy generated code) or the
20 // OSSS+R model.
21 #if USE_SYNTHESISED_DUT
22 #include "synthesised_dispatchModule.cc"
23 #else
24 #include "dispatch_dut.cc"
25 #endif
26
27 #ifdef SC_SYNTHESIS
28 // do nothing
29 #define TEST_CALL( a, b )
30 #else
31 #define TEST_CALL( methodName, testcode )\
32   expected_id = b_ptr->methodName();\
33   calculated_id = check(testcode);\
34   if (expected_id == calculated_id)\
35   { logfile <<  "OK:␣␣␣" << #methodName\
36             << "␣=␣" << showCode(calculated_id)\
37             << std::endl; }\
38   else\
39   { logfile << "FAIL:␣" << #methodName\
40             << "␣=␣" << showCode(calculated_id)\
41             << ",␣expected␣=␣" << showCode(expected_id)\
42             << std::endl; }
43 #endif
44
45
46
47
```

```
48 #ifdef SC_SYNTHESIS
49 CODEC_ID_TYPE showCode( CODEC_ID_TYPE x) { return x; };
50 #else
51
52 #include <iostream>
53
54 std::string showCodePart(int part)
55 {
56   std::string buf;
57   switch ( part )
58   {
59     case V_MARK : buf = "␣V␣"; break;
60     case N_MARK : buf = "␣N␣"; break;
61     case P_MARK : buf = "␣P␣"; break;
62     case M_MARK : buf = "␣M␣"; break;
63     default:      buf = "ERR"; break;
64   }
65   return buf;
66 }
67
68 std::string
69 showCode(CODEC_ID_TYPE testcode)
70 {
71   std::string buf;
72   switch ( classMark( testcode.to_int() ) )
73   {
74     case A_MARK  : buf = "class␣A␣"; break;
75     case B_MARK  : buf = "class␣B␣"; break;
76     case C_MARK  : buf = "class␣C␣"; break;
77     case NO_MARK : buf = "no␣class"; break;
78     default:       buf = "ERR"     ; break;
79   }
80   buf = buf + "/" + showCodePart(aMark(testcode.to_int()))
81             + "/" + showCodePart(bMark(testcode.to_int()))
82             + "/" + showCodePart(cMark(testcode.to_int()));
83   return buf;
84 }
85
86 #endif // SC_SYNTHESIS
87
88 SC_MODULE(tbModule)
89 {
90   sc_out< CODEC_ID_TYPE >   po_method_id;
91   sc_out< bool >            po_req;
92   sc_in< CODEC_ID_TYPE >    pi_method_id;
93   sc_in< bool >             pi_ack;
94
95   sc_in< bool >   pi_clock;
96   sc_out< bool >  po_reset;
97   sc_signal< bool > s_reset;
98
99   B * b_ptr;
100
101  // This method performs one method invocation test
102  CODEC_ID_TYPE
103  check(CODEC_ID_TYPE testcode)
104  {
105    po_method_id.write( testcode );
106    po_req.write( true );
107    wait();
108    while (pi_ack.read() == false){ wait(); }
109    po_req.write(false);
110    CODEC_ID_TYPE calculated_id = pi_method_id.read();
111    wait();
112    while (pi_ack.read() == true){ wait(); }
113    return calculated_id;
114  }
115
```

```
116   void tester()
117   {
118     po_req.write(false);
119     wait();
120
121     CODEC_ID_TYPE expected_id;
122     CODEC_ID_TYPE calculated_id;
123
124     b_ptr = new C();
125
126     TEST_CALL(m_vvv, code(A_MARK, V_MARK, V_MARK, V_MARK))
127     TEST_CALL(m_vvn, code(A_MARK, V_MARK, V_MARK, N_MARK))
128     TEST_CALL(m_vvm, code(A_MARK, V_MARK, V_MARK, M_MARK))
129     TEST_CALL(m_vnv, code(A_MARK, V_MARK, N_MARK, V_MARK))
130     TEST_CALL(m_vnn, code(A_MARK, V_MARK, N_MARK, N_MARK))
131     TEST_CALL(m_vnm, code(A_MARK, V_MARK, N_MARK, M_MARK))
132     TEST_CALL(m_vmv, code(A_MARK, V_MARK, M_MARK, V_MARK))
133     TEST_CALL(m_vmn, code(A_MARK, V_MARK, M_MARK, N_MARK))
134     TEST_CALL(m_vmm, code(A_MARK, V_MARK, M_MARK, M_MARK))
135
136     TEST_CALL(m_nvv, code(A_MARK, N_MARK, V_MARK, V_MARK))
137     TEST_CALL(m_nvn, code(A_MARK, N_MARK, V_MARK, N_MARK))
138     TEST_CALL(m_nvm, code(A_MARK, N_MARK, V_MARK, M_MARK))
139     TEST_CALL(m_nnv, code(A_MARK, N_MARK, N_MARK, V_MARK))
140     TEST_CALL(m_nnn, code(A_MARK, N_MARK, N_MARK, N_MARK))
141     TEST_CALL(m_nnm, code(A_MARK, N_MARK, N_MARK, M_MARK))
142     TEST_CALL(m_nmv, code(A_MARK, N_MARK, M_MARK, V_MARK))
143     TEST_CALL(m_nmn, code(A_MARK, N_MARK, M_MARK, N_MARK))
144     TEST_CALL(m_nmm, code(A_MARK, N_MARK, M_MARK, M_MARK))
145
146     /* Methods that are pure virtual in A cannot be
147        invoked on A.
148     TEST_CALL(m_pvv, code(A_MARK, P_MARK, V_MARK, V_MARK))
149     TEST_CALL(m_pvn, code(A_MARK, P_MARK, V_MARK, N_MARK))
150     TEST_CALL(m_pvm, code(A_MARK, P_MARK, V_MARK, M_MARK))
151     TEST_CALL(m_pnv, code(A_MARK, P_MARK, N_MARK, V_MARK))
152     TEST_CALL(m_pnn, code(A_MARK, P_MARK, N_MARK, N_MARK))
153     TEST_CALL(m_pnm, code(A_MARK, P_MARK, N_MARK, M_MARK))
154     */
155
156     TEST_CALL(m_mvv, code(A_MARK, M_MARK, V_MARK, V_MARK))
157     TEST_CALL(m_mvn, code(A_MARK, M_MARK, V_MARK, N_MARK))
158     TEST_CALL(m_mvm, code(A_MARK, M_MARK, V_MARK, M_MARK))
159     TEST_CALL(m_mnv, code(A_MARK, M_MARK, N_MARK, V_MARK))
160     TEST_CALL(m_mnn, code(A_MARK, M_MARK, N_MARK, N_MARK))
161     TEST_CALL(m_mnm, code(A_MARK, M_MARK, N_MARK, M_MARK))
162
163 #ifndef SC_SYNTHESIS
164     sc_stop();
165 #endif
166   }
167
168
169   void resetGenerator()
170   {
171     OSSS_MESSAGE(VERBOSE_DEBUG_TB_MODE, "Reset␣start");
172     s_reset.write( true );
173     wait(2);
174
175     OSSS_MESSAGE(VERBOSE_DEBUG_TB_MODE, "Reset␣end");
176     s_reset.write( false );
177     for (;;) wait(1000);
178   }
179
180   void resetDriver()
181   {
182     po_reset.write( s_reset );
183   }
```

```
184
185   SC_CTOR(tbModule) : po_method_id("po_method_id"),
186                       po_req("po_req"),
187                       pi_method_id("pi_method_id"),
188                       pi_ack("pi_ack"),
189                       po_reset("po_reset"),
190                       pi_clock("pi_clock")
191   {
192     SC_CTHREAD(tester, pi_clock.pos());
193     reset_signal_is(s_reset, true);
194
195     SC_CTHREAD(resetGenerator, pi_clock.pos());
196     // no reset input!
197
198     SC_METHOD( resetDriver );
199     sensitive << s_reset;
200   }
201 };
202
203 int sc_main(int /*argc*/, char * /*argv*/[])
204 {
205   // First the well-known ordinary SystemC-stuff...
206 #ifndef SC_SYNTHESIS
207   sc_clock clock("main_clock", sc_time(10, SC_NS));
208   sc_signal< bool > reset("reset");
209 #endif
210   dispatchModule dut("design_under_test");
211   tbModule tb("testbench_module");
212
213   sc_signal< CODEC_ID_TYPE >  tb2dut_method_id("tb2dut_method_id");
214   sc_signal< CODEC_ID_TYPE >  dut2tb_method_id("dut2tb_method_id");
215   sc_signal< bool > tb2dut_req("tb2dut_go");
216   sc_signal< bool > dut2tb_ack("dut2tb_ok");
217
218   dut.pi_method_id( tb2dut_method_id );
219   dut.pi_req(       tb2dut_req );
220   dut.po_method_id( dut2tb_method_id );
221   dut.po_ack(       dut2tb_ack );
222
223   tb.po_method_id( tb2dut_method_id );
224   tb.po_req(       tb2dut_req );
225   tb.pi_method_id( dut2tb_method_id );
226   tb.pi_ack(       dut2tb_ack );
227
228 #ifndef SC_SYNTHESIS
229   dut.clock(clock);
230   dut.reset(reset);
231   tb.pi_clock(clock);
232   tb.po_reset(reset);
233 #endif
234
235 #if not USE_SYNTHESISED_DUT
236   // ... and here are the OSSS+R statements:
237
238   // We declare that we have a special type of FPGA. This is
239   // a type declaration (like "Xilinx Virtex II 1000-4").
240   osss::osss_device_type my_device_type("My FPGA device type");
241
242   // Specify, how the logic copy times (unused in this example,
243   // so please ignore the 26us entries) and reconfiguration times
244   // are, if a certain class (e.g. OggVorbisPlayer<Memory> is to
245   // be configured to the FPGA.
246   OSSS_DECLARE_TIME(my_device_type, A, sc_time(1, SC_US), sc_time(100, SC_US));
247   OSSS_DECLARE_TIME(my_device_type, B, sc_time(1, SC_US), sc_time(100, SC_US));
248   OSSS_DECLARE_TIME(my_device_type, C, sc_time(1, SC_US), sc_time(100, SC_US));
249   osss::osss_device my_device(my_device_type, "my_device");
250
251   dut.ro( my_device );
```

```
252 #else
253   //rctrl_my_device_module my_device("my_synthesised_fpga");
254 #endif
255
256   // The rest is ordinary SystemC stuff again
257 #ifndef SC_SYNTHESIS
258 #if not USE_SYNTHESISED_DUT
259   my_device.clock_port(clock);
260   my_device.reset_port(reset);
261 #else // USE_SYNTHESISED_DUT == true :
262   // This is a workaround: In future, fossy should generate
263   // the device module as its own top-level instance.
264   dut.my_device.clock_port(clock);
265   dut.my_device.reset_port(reset);
266
267   /* No longer required on single-up RO
268   // emulate fake crossbar
269   sc_signal< sc_bigint< 32 > > cb_upi("cb_upi");
270   dut.ro.cbi_user_process( cb_upi );
271   */
272 #endif // else USE_SYNTHESISED_DUT
273 #endif // not SC_SYNTHESIS
274
275
276   OSSS_MESSAGE(VERBOSE_DEBUG_TB_MODE, "Starting simulation");
277 #ifndef SC_SYNTHESIS
278 #if USE_SYNTHESISED_DUT
279   logfile.open ("simulation_results_synth.txt");
280 #else
281   logfile.open ("simulation_results_plusr.txt");
282 #endif
283 #endif
284
285 #ifndef SC_SYNTHESIS
286   sc_start();
287 #endif
288   OSSS_MESSAGE(VERBOSE_DEBUG_TB_MODE, "Stopped simulation");
289   return EXIT_SUCCESS;
290 }
```

Bibliography

[1] *ANDRES project website*. http://andres.offis.de/.

[2] Edison design group website. http://www.edg.com/.

[3] *Haskell Website*. http://www.haskell.org/.

[4] *Object Management Group Website*. http://www.uml.org.

[5] *ODETTE project website*. http://odette.offis.de.

[6] *Perl Website*. http://www.perl.org.

[7] Audio Codec '97. Technical Report Rev. 2.3, Intel Corporation, Apr. 2002.

[8] Field update fpgas while system operates. Technical report, Lattice Semiconductor, 555 Northeast More Ct., Hillsboro, Oregon 97124 USA, May 2005.

[9] *ModelSim Datasheet*. http://www.mentor.com/products/fv/digital_verification/modelsim_se, mgc 5378:080709 edition, 2008.

[10] Ali Ahmadinia, Christophe Bobda, Dirk Koch, Mateusz Majer, and Jürgen Teich. Task Scheduling for Heterogeneous Reconfigurable Computers. *17th Symposium on Integrated Circuits and Systems Design*, Sept. 2004.

[11] Carsten Albrecht, Roman Koch, Thilo Pionteck, and Erik Maehle. Simulation System for Run-Time Reconfigurable Networks-on-Chip. In *Proceedings of the 6th EUROSIM Congress on Modelling and Simulation*, Sept. 2007.

[12] Carsten Albrecht, Thilo Pionteck, Roman Koch, and Erik Maehle. Modelling Tile-Based Run-Time Reconfigurable Systems Using SystemC. In *Proceedings of the European Conference on Modelling and Simulation 2007*, pages 509–514, 2007.

[13] Altera Corp., http://www.altera.com/literature/hb/cyc3/cyc3_ciii52001.pdf. *Cyclone III Device Handbook*, ciii52001-2.2 edition, Oct. 2008.

[14] Hideharu Amano, Akiya Jouraku, and Kenichiro Anjo. A Dynamically Adaptive Hardware on Dynamically Reconfigurable Processor. *IEICE Transactions on Communications E Series B*, 86(1):3385–3391, Jan. 2003.

[15] C. Amicucci, F. Ferrandi, M. Santambrogio, and D. Sciuto. SyCERS: a SystemC design exploration framework for SoC reconfigurable architecture. *International Conference on Engineering of Reconfigurable Systems & Algorithm (ERSA)*, pages 63–69, June 2006.

[16] Kenji Asano, Junji Kitamichi, and Kenchi Kuroda. Proposal of Dynamic Module Library for System Level Modeling and Simulation of Dynamically Reconfigurable Systems. *20th International Conference on VLSI Design (VLSID)*, pages 373–378, Jan. 2007.

[17] Atmel Corp., `http://www.atmel.com/dyn/resources/prod_documents/DOC0896.PDF`. *AT40K05/10/20/40(LV) Mature*, May 2002.

[18] Atmel Corp., `http://www.atmel.com/dyn/resources/prod_documents/doc1138.pdf`. *AT94KAL Series Field Programmable System Level Integrated Circuit*, Jan. 2008.

[19] Salih Bayar and Arda Yurdakul. Dynamic Partial Self-Reconfiguration on Spartan-III FPGAs via a Parallel Configuration Access Port (PCAP). In *Proceedings of HiPEAC Workshop on Reconfigurable Computing*, Jan. 2008.

[20] Peter Bellows and Brad Hutchings. JHDL – An HDL for Reconfigurable Systems. *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, page 175, Apr. 1998.

[21] Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. Lava: Hardware Design in Haskell. In *Proceedings of the third ACM SIGPLAN International Conference on Functional Programming*, pages 174–184. ACM Press, Jan. 1999.

[22] Christophe Bobda, Ali Ahmadinia, Kurapati Rajesham, and Mateusz Majer. Partial Configuration Design and Implementation Challenges on Xilinx Virtex FPGAs. *System Aspects in Organic and Pervasive Computing - Workshop Proceedings - Dynamically Reconfigurable Systems, Self-Organization and Emergence (DRS-ARCS).*, pages 61–66, Mar. 2005.

[23] Christophe Bobda, Mateusz Majer, Ali Ahmadinia, Thomas Haller, Andre Linarth, Jurgen Teich, and Jan van der Veen. The Erlangen Slot Machine: A Highly Flexible FPGA-Based Reconfigurable Platform. In *FCCM '05: Proceedings of the 13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 319–320, Washington, DC, USA, 2005. IEEE Computer Society.

[24] Kiran Bondalapati and Viktor K. Prasanna. Reconfigurable Computing Systems. *Proceedings of the IEEE*, 90(7):1201–1217, 2002.

[25] Grady Booch. *Object-Oriented Analysis and Design with Applications (3rd Edition)*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 2004.

[26] Alisson V. Brito, Matthias Kuhnle, Michael Hubner, Jurgen Becker, and Elmar U. K. Melcher. Modelling and Simulation of Dynamic and Partially Reconfigurable Systems using SystemC. *ISVLSI '07: Proceedings of the IEEE Computer Society Annual Symposium on VLSI*, pages 35–40, Mar. 2007.

[27] Alisson V. Brito, Elmar U.K. Melcher, and Wilson Rosas. An open-source tool for simulation of partially reconfigurable systems using SystemC. *Emerging VLSI Technologies and Architectures (ISVLSI)*, 2006.

[28] Claus Brunzema, Cornelia Grabbe, Kim Grüttner, Philipp Andreas Hartmann, Andreas Herrholz, Henning Kleen, Frank Oppenheimer, Andreas Schallenberg, Christian Stehno, and Thorsten Schubert. *OSSS - A Library for Synthesisable System Level Models in SystemC(TM) - A tutorial for OSSS 2.0*, 2007.

[29] Claus Brunzema and Frank Oppenheimer. Integer Data Type Semantics: SystemC vs. VHDL. Technical report, OFFIS, R6D Division Transportation, www.offis.de/publikationen/download.php?id=002055, 09 2008.

[30] Enrique Canto, Mariano Lopez, and Francesc Fons. Self-Reconfiguration of Embedded Systems Mapped on Spartan-3. *Procceedings of Reconfigurable Communication-centric SoCs*, pages 117–124, July 2008.

[31] Katherine Compton and Scott Hauck. Totem: Custom Reconfigurable Array Generation. In *The 9th Annual IEEE Symposium on FPGAs for Custom Computing Machines*, pages 111–119, Apr.-May 2001.

[32] Katherine Compton and Scott Hauck. Reconfigurable Computing: A survey of Systems and Software. In *ACM Computing Surveys*, volume 34. Northwestern University and University of Washington, ACM Press, Jun. 2002.

[33] Andreas Dandalis and Viktor K. Prasanna. Configuration Compression for FPGA-Based Embedded Systems. In *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, volume 13, pages 1394–1398, Dec. 2005.

[34] Klaus Danne, Roland Mühlenbernd, and Marco Platzner. Executing Hardware Tasks on Dynamically Reconfigurable Devices under Real-Time Conditions. *Procceedings of Field-Programmable Logic (FPL)*, pages 1–6, 2006.

[35] Jean Philippe Delahaye, Guy Gogniat, Christian Roland, and Pierre Bomel. Software Radio and Dynamic Reconfiguration on a DSP/FPGA platform. *Frequenz - Journal of Telecommunications*, (58):152–159, May-June 2004.

[36] Giovanni Denaro and Leonardo Mariani. Towards Testing and Analysis of Systems that Use Serialization. In *International Workshop on Test and Analysis of Component Based Systems (TACoS) satellite workshop at the European Joint Conferences on Theory and Practice of Software (ETAPS)*, volume 116 of *ENTCS*, pages 171–184. Elsevier, 2004.

[37] Arran Derbyshire, Tobias Becker, and Wayne Luk. Incremental elaboration for run-time reconfigurable hardware designs. In *Proceedings of the 2006 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, pages 93–102, Oct. 2006.

[38] Arran Derbyshire and Wayne Luk. Compiling Run-Time Parameterisable Designs. *Procceedings of IEEE International Conference on Field-Programmable Technology*, pages 44–51, Sept. 2002.

[39] Florian Dittmann. PART-E. Design and Automation Conference in Europe, University Booth, Mar. 2006.

[40] Florian Dittmann. PART-E project website. Technical report, University of Paderborn, http://sourceforge.net/projects/parte/, 2007.

[41] Florian Dittmann and Christophe Bobda. Temporal placement on mesh-based coarse grain reconfigurable systems using the spectral method. In *From Spec. to Embedded Sys. Application, Proc. of the IESS*, pages 301–310. Kluwer, Aug 2005.

[42] Alberto Donato, Fabrizio Ferrandi, Massimo Redaelli, Marco D. Santambrogi, and Donatella Sciuto. Caronte: a complete methodology for the implementation of partially dynamically self-reconfigurable systems on FPGA platforms. In *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'05)*, pages 321–322. IEEE Computer Society, 2005.

[43] Thomas Eisenbarth, Christof Paar, Axel Poschmann, Sandeep Kumar, and Leif Uhsadel. A Survey of Lightweight Cryptography Implementations. *IEEE Design and Test of Computers, Design and Test of ICs for Secure Embedded Computing*, pages 522–533, Nov.-Dec. 2007.

[44] Alexandra Poetter et al. JHDLBits: The Merging of Two Worlds. In *Proceedings of the 14th International Workshop on Field-Programmable Logic and Applications*, Aug. 2004.

[45] Emi Eto. *XAPP290 Difference-Based Partial Reconfiguration*. `http://www.xilinx.com/support/documentation/application_notes/xapp290.pdf`, v2.0 edition, Dec. 2007.

[46] Sandor P. Fekete, E. Köhler, and Jürgen Teich. Optimal FPGA module placement with temporal precedence constraints. In *Proc. DATE 2001, Design, Automation and Test in Europe*, pages 658–665, Mar 2001.

[47] FOSSY *synthesiser*. `http://fossy.offis.de`.

[48] Kris Gaj and Pawel Chodowiec. Fast implementation and fair comparison of the final candidates for Advanced Encryption Standard using Field Programmable Gate Arrays. In *Proceedings of the RSA Security Conference - Cryptographer's Track*, Apr. 2001.

[49] Philipp Garcia, Katherine Compton, Michael Schulte, Emily Blem, and Wenyin Fu. An Overview of Reconfigurable Hardware in Embedded Systems. In *EURASIP Journal of Embedded Systems*, number 1, pages 1–19, Jan. 2006.

[50] Soheil Ghiasi and Majid Sarrafzadeh. An Optimal Algorithm for Minimizing Runtime Reconfiguration Delay. *ACM Transactions on Embedded Computing Systems*, 3:237–256, May 2004.

[51] Ralph Görgen, Frank Oppenheimer, Andreas Schallenberg, and Wolfgang Nebel. Analyse und optimierung von dynamisch rekonfigurierbaren systemen mittels ereignisvisualisierung. In *Tagungsband des 11. ITG/GMM/GI-Workshop "Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen"*, Mar. 2008.

[52] Eike Grimpe. *Performance Optimising Hardware Synthesis of Shared Objects*. PhD thesis, Universität Oldenburg, `http://www.offis.de/publikationen`, 2005.

[53] Eike Grimpe, Bernd Timmermann, Tiemo Fandrey, Ramon Biniasch, and Frank Oppenheimer. SystemC Object-Oriented Extensions and Synthesis Features. In *Proceedings - Forum on Design and Specification Languages FDL '02*, Sept. 2002.

[54] Reactive Systems Group. *The Quartz Language Reference Manual 1.8*. University of Kaiserslautern, May 2006.

[55] Kim Grüttner, Claus Brunzema, Cornelia Grabbe, Thorsten Schubert, and Frank Oppenheimer. OSSS-Channels: Modelling and Synthesis of Communication with SystemC. In *Proceedings of the Forum on Specification and Design Languages, FDL'06*, Sept. 2006.

[56] Steve Guccione, Delon Levi, and Prasanna Sundararajan. JBits: Java based interface for reconfigurable computing. *2nd Annual Military and Aerospace Applications of Programmable Devices and Technologies (MAPLD)*, Sept. 1999.

[57] Reiner Hartenstein. A Decade of Reconfigurable Computing: a Visionary Retrospective. *Design and Automation Conference in Europe*, page Embedded Tutorial, 2001.

[58] Yohei Hori, Akashi Satoh, Hirofumi Sakane, and Kenji Toda. Bitstream encryption and authentication with AES-GCM in dynamically reconfigurable systems. In *International Conference on Field Programmable Logic and Applications (FPL'2008)*, pages 23–28, Sept. 2008.

[59] Pao-Ann Hsiung, Chun-Hsian Huang, and Chih-Feng Liao. Perfecto: A SystemC-Based Performance Evaluation Framework for Dynamically Partially Reconfiguable Systems. *Procceedings of Field-Programmable Logic and Applications (FPL)*, pages 190–195, Aug. 2006.

[60] Brad Hutchings, Peter Bellows, Joseph Hawkins, et al. A CAD Suite for High-Performance FPGA Design. *IEEE Symposium on Field-Programmable Custom Computing Machines*, 0:12, 1999.

[61] Heiko Kalte and Mario Porrmann. Context Saving and Restoring for Multitasking in Reconfigurable Systems. *Procceedings of Field-Programmable Logic (FPL)*, pages 223–228, Aug. 2005.

[62] Cindy Kao. Benefits of Partial Reconfiguration. *Xcell Journal*, (55):65–67, Q4 2005.

[63] Junji Kitamichi, Koji Ueda, and Kenichi Kuroda. A Modelling of a Dynamically Reconfigurable Processor using SystemC. *21st International Conference on VLSI Design*, pages 91 – 96, Jan. 2008.

[64] L. Kriaa, S. Adriano, E. Vaumorin, R. Nouacer, F. Blanc, , S. Pajaniardja, P. Coussy, E. Martin, , D. Heller, F. Tabet, and A.M. Fouilliart. SystemC'mantic: A high level Modeling and Co-design Framework For Reconfigurable Real Time Systems. *Proceedings - Forum on Specification and Design Languages, FDL'05*, Sept. 2005.

[65] Lattice Semiconductor Corp., `http://www.latticesemi.com/products/fpga/xp/index.cfm`. *LatticeXP Family Handbook*, hb1001 v.03.1 edition, Nov. 2007.

[66] Edward A. Lee. Overview of the Ptolemy Project. Technical report, University of California, Jul. 2003. Technical Memorandum No. UCB/ERL M03/25.

[67] Gareth Lee and George Milne. A methodology for design of run-time reconfigurable systems. *IEEE International Conf. on Field-Programmable Technology (FPT)*, pages 60–67, Dec. 2002.

[68] Gareth Lee and George Milne. Building Run-time Reconfigurable Systems from Tiles277. In *Field-programmable Logic and Applications (FPL)*, volume 2778 of *LNCS*, pages 252–261. Springer-Verlag Berlin Heidelberg, 2003.

[69] T.K. Lee, A. Derbyshire, W. Luk, and Peter Y. K. Cheung. High-level language extensions for run-time reconfigurable systems. In *Proceedings of IEEE International Conference on Field-Programmable Technology (FPT)*, pages 144–151, Dec. 2003.

[70] Jason Leonard and William H. Mangione-Smith. A Case Study of Partially Evaluated Hardware Circuits: Key-Specific DES. In Wayne Luk, Peter Y. K. Cheung, and Manfred Glesner, editors, *Field-Programmable Logic and Applications. 7th International Workshop*, volume 1304, pages 151–160, London, U.K., 1997. Springer-Verlag.

[71] Timothy C. Lethbridge and Robert Laganiere. *Object-Oriented Software Engineering*. McGraw-Hill Education, Shoppenhangers Road, Maidenhead, Berkshire, SL6 2QL, England, 2001.

[72] Barbara H. Liskov and Jeannette M. Wing. Behavioral Subtyping Using Invariants and Constraints. Technical report, Carnegie-Mellon University Pittsburgh, Dept. of Computer Science, July 1999.

[73] Wayne Luk and Steve McKeever. Pebble: A language for parametrised and reconfigurable hardware design. In *Proceedings - 1998 International Conference on Field Programmable Logic and Applications*, pages 9–18, Aug.-Sept. 1998.

[74] Wayne Luk, Nabeel Shirazi, and Peter Y. K. Cheung. Modelling and Optimising Run-Time Reconfiguable Systems. *Procceedings of IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 167–176, Apr. 1996.

[75] Wayne Luk, Nabeel Shirazi, and Peter Y. K. Cheung. Compilation tools for run-time reconfigurable designs. In *Procceedings IEEE Symposium on Field-Programmable Custom Computin Machines (FCCM)*, pages 56–65. IEEE Computer Society, 1997.

[76] Patrick Lysaght, Brandon Blodget, Jeff Mason, Jay Young, and Brendan Bridgford. Enhanced Architectures, Design Methodologies and CAD Tools for Dynamic Reconfiguration of Xilinx FPGAs. *International Conference on Field Programmable Logic and Applications*, pages 1–6, Aug. 2006.

[77] Fred Ma, John P. Knight, and Calvin Plett. Physical Resource Binding for a Coarse-Grain Reconfigurable Array Using Evolutionary Algorithms. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 13(5):553–563, May 2005.

[78] Konstantinos Masselos, Nikolaos S. Voros, Yang Qu, Kari Tiensyrjä, Miroslav Cupák, Luc Rijnders, and Marko Pettissalo. System Level Architecture Exploration for Reconfigurable Systems on Chip. *Procceedings of Field-Programmable Logic (FPL'06)*, pages 59–64, 2006.

[79] Bingfeng Mei, Andy Lambrechts, Jean-Yves Mignolet, Diederik Verkest, and Rudy Lauwereins. Architecture Exploration for a Reconfigurable Architecture Template. *IEEE Design and Test of Computers*, pages 90–101, Mar.-Apr. 2005.

[80] Oskar Mencer, Luc Séméria, Martin Morf, and Jean-Marc Delosme. Application of Reconfigurable CORDIC Architectures. *Journal of VLSI Signal Processing Systems*, 24(2-3):211–221, 2000.

[81] Gregory Mermoud. A Module-Based Dynamic Partial Reconfiguration tutorial. `http://www.tc.umn.edu/~kimx0746/ref_papers/DPRtutorial.pdf`, Nov. 2004.

[82] MIPS Technologies, `http://www.mips.com`. *Accelerating VoIP with CorExtend in MIPS32 Pro Series Cores*, Jan. 2003.

[83] Luciano Musa. FPGAs in High Energy Physics Experiments at CERN. In *Procceedings of Field-Programmable Logic (FPL)*, page 2, Sept. 2008.

[84] Kazuteru Namba and Hideo Ito. Proposal of Testable Multi-Context FPGA Architecture. *IEICE - Trans. Inf. Syst.*, E89-D(5):1687–1693, 2006.

[85] Kelly Nasi, Theodore Karoubalis, Martin Daněk, and Zdeněk Pohl. FIGARO - An Automatic Tool Flow for Designs with Dynamic Reconfiguration. *Proceedings - International Conference on Field Programmable Logic and Applications (FPL'05)*, pages 590–593, Aug. 2005.

[86] Stephen Neuendorffer. Automatic Specialization of Actor-oriented Models in Ptolemy II. Technical memorandum ucb erl m02/41, Electronics Research Laboratory, University of California at Berkeley, Dec. 2002.

[87] Juanjo Noguera and Rosa M. Badia. Run-Time HW/SW Codesign for Discrete Event Systems Using Dynamically Reconfigurable Architectures. In *ISSS '00: Proceedings of the 13th International Symposium on System Synthesis*, pages 100–106, Washington, DC, USA, 2000. IEEE Computer Society.

[88] Juanjo Noguera and Rosa M. Badia. HW/SW Codesign Techniques for Dynamically Reconfigurable Architectures. In *IEEE Transactions on Very Large Scale Integration (VLSI) Systems, Vol. 10, No. 4*, pages pp 399–415. IEEE, Aug. 2002.

[89] Open SystemC Initiative. *SystemC*™. `http://www.systemc.org`.

[90] Open SystemC Initiative, 1177 Braham Lane 302, San Jose, CA 95118-3799. *SystemC 2.0.1 Language Reference Manual*, revision 1.0 edition, 2003. `http://www.systemc.org`.

[91] PACT Informationstechnologie GmbH, `http://www.pactxpp.com`. *The XPP White Paper - A Technical Perspective*, Mar. 2002.

[92] Antti Pelkonen, Kostas Masselos, and Miroslav Cupák. System-Level Modeling of Dynamically Reconfigurable Hardware with SystemC. *Proceedings of International Symposium on Parallel and Distributed Processing (Reconfigurable Architectures Workshop)*, pages 174–181, Apr. 2003.

[93] Oliver Pell and Wayne Luk. Quartz: A Framework for Correct and Efficient Reconfigurable Design. *Proceedings of the International Conference on Reconfigurable Computing and FPGAs (ReConFig)*, pages 8–14, Sept. 2005.

[94] Oliver Pell and Wayne Luk. Compiling Higher-Order Polymorphic Hardware Descriptions into Parameterised VHDL Libraries with Flexible Placement Information. *Procceedings of Field-Programmable Logic and Applications (FPL)*, pages 125–130, Aug. 2006.

[95] Monica Magalães Pereira, Bruno Cruz de Oliveira, and Ivan Saraiva Silva. RoSA: a Reconfigurable Stream-based Architecture. *20th Symposium on Integrated Circuits and Systems Design*, pages 159–164, Sept. 2007.

[96] Xiao ping Ling and Hideharu Amano. Performance evaluation of WASMII: a data driven computer on a virtual hardware. In *Proceedings. IEEE Workshop on FPGAs for Custom Computing Machines*, pages 33–42, Apr. 1993.

[97] Thilo Piontek, Carsten Albrecht, Roman Koch, Torben Brix, and Erik Maehle. Design and Simulation of Runtime Reconfigurable Systems. In *IEEE Workshop on Design and Diagnostics of Electronic Circuits and Systems (DDECS 2008)*, pages 154–157. IEEE, 2008.

[98] Alexandra Vanessa Poetter. JHDLBits: An Open-Source model for FPGA Design Automation. Master's thesis, Virginia Polytechnic Institute and State University, Blacksburg, Virginia, Aug. 2004.

[99] Adriatic Project. *Final Report*, Sept. 2004. IST-2000-30049/WP6/FR/ICOM/R.

[100] Uwe Proß, Benjamin Berger, Ulrich Heinkel, and Dietmar Müller. Virtueller Prototyp für dynamisch partiell rekonfigurierbare Systeme. *Dresdner Arbeitstagung Schaltungs- und Systementwurf*, pages 91–96, May 2006. German.

[101] Kiran Puttegowds et al. Context Switching in a Run-Time Reconfigurable System. *The Journal of Supercomputing*, 26:239–257, 2003.

[102] Yang Qu, Juha-Pekka Soininen, and Jari Nurmi. Improving the Efficiency of Run Time Reconfigurable Devices by Configuration Locking. *Proceedings of Design, Automation and Test in Europe*, pages 264–267, march 2008.

[103] Andreas Raabe, Philipp A. Hartmann, and Joachim K. Anlauf. ReChannel: Describing and Simulating Reconfigurable Hardware in SystemC. In *ACM Transactions on Design Automation of Electronic Systems*, volume 13, Jan. 2008.

[104] Andreas Raabe, Andreas Nett, and Andreas H. Niers. A Refinement Case-Study of a Dynamically Reconfigurable Intersection Test Hardware. *Proceedings of Reconfigurable Communication-centric SoCs (ReCoSoC)*, July 2008.

[105] Javier Resano, Daniel Mozos, Francky Catthoor, and Diederik Verkest. A Reconfiguration Manager for Dynamically Reconfigurable Hardware. *IEEE Design and Test of Computers*, pages 452–460, Sept.-Oct. 2005.

[106] Tero Rissa, Milan Vasilko, and Jarkko Niittylathi. System-Level Modelling and Implementation Technique for Run-Time Reconfigurable Systems. *Procceedings of the 10th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 295–296, 2002.

[107] Ian Robertson and James Irvine. A design flow for partially reconfigurable hardware. *Trans. on Embedded Computing Sys.*, 3(2):257–283, May 2004.

[108] Ian Robertson, James Irvine, Patrick Lysaght, and David Robinson. Improved functional simulation of dynamically reconfigurable logic. *Lecture Notes in Computer Science*, 2438:152–161, 2002. ISSN 0302-9743.

[109] Ian Robertson, James Irvine, Patrick Lysaght, and David Robinson. Timing Verification of Dynamically Reconfigurable Logic for the Xilinx Virtex FPGA Series. *Proceedings of the ACM/SIGDA tenth International Symposium on Field Programmable Gate Arrays*, pages 127–135, Feb. 2002.

[110] David Robinson. *Simulation and Control of Dynamically Reconfigurable Logic Circuits*. PhD thesis, University of Strathclyde, UK, 2002. PhD thesis.

[111] David Robinson and Patrick Lysaght. Modelling and Synthesis of Configurable Controllers for Dynamically Reconfigurable Logic Systems Using the DCS CAD Framework. *Procceedings of Field-Programmable Logic and Applications (FPL)*, pages 41–50, Sept. 1999.

[112] David Robinson and Patrick Lysaght. Methods of exploiting simulation technology for simulating the timing of dynamically reconfigurable logic. *IEE proceedings. Computers and digital techniques*, 147:175–180, 2000. ISSN 1350-2387.

[113] David Robinson and Patrick Lysaght. Verification of Dynamically Reconfigurable Logic. *Lecture Notes In Computer Science; Proceedings of the The Roadmap to Reconfigurable Computing, 10th International Workshop on Field-Programmable Logic and Applications*, 1896:141–150, 2000. ISBN:3-540-67899-9.

[114] David Robinson, Gordon McGregor, and Patrick Lysaght. New CAD Framework Extends Simulation of Dynamically Reconfigurable Logic. *Procceedings of Field Programmable Logic and Applications*, pages 1–8, Sept. 1998.

[115] Marco D. Santambrogio. The Caronte Approach for Dynamically Reconfigurable Systems. *PhD Forum, 14th International Conference on Very Large Scale Integration (VLSI-SoC)*, Oct. 2006.

[116] Jorge Scandaliaris, Juan M. Moreno, Joan Cabestany, Philippe Buttel, Alain Rachet, Jiri Kadlec, Antonin Hermanek, Dominic de Saint Roman, Gerard Habay, and Alessandro Donati. A General Design Flow for Dynamically Reconfiguarble FPGAs (D_FPGAs). *Reconfigurable Architectures Workshop*, Apr. 2003.

[117] Patrick Schaumont, Ingrid Verbauwhede, Kurt Keutzer, and Majid Sarrafzadeh. A Quick Safati Through the Reconfiguration Jungle. *Procceedings of the 38th Design Automation Conference*, pages 172–177, June 2001.

[118] Bruce Schneier. Description of a New Variable-Length Key, 64-Bit Block Cipher (Blowfish). *Lecture Notes in Computer Science*, 809:191–204, 1994.

[119] Nabeel Shirazi, Wayne Luk, and Peter Y. K. Cheung. Run-Time Management of Dynamically Reconfigurable Designs. *Procceedings of Field-Programmable Logic (FPL'98)*, pages 59–68, Aug. 1998.

[120] Hartej Singh, Ming-Hau Lee, Guangming Lu, Fadi J. Kurdahi, Nader Bagherzadeh, and Eliseu M.C. Filho. MorphoSys: an integrated reconfigurable system for data-parallel and computation-intensive applications. In *IEEE Transactions on Computers*, volume 49, pages 465–481, May 2000.

[121] S. Singh, P. James-Roxby, and X. Inc. Lava and JBits: From HDL to Bitstream in Seconds. In *The 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'01)*, pages 91–100, Apr.-May 2001.

[122] Satnam Singh. Designing Reconfigurable Systems in Lava. In *VLSID '04: Proceedings of the 17th International Conference on VLSI Design*, page 299, Washington, DC, USA, 2004. IEEE Computer Society.

[123] Christoph Steiger, Herbert Walder, and Marco Platzner. Operating Systems for Reconfigurable Embedded Platforms: Online Scheduling of Real-Time Tasks. *IEEE Transactions on Computers*, 53(11):1393–1407, Nov. 2004.

[124] Jon Stockwood and Patrick Lysaght. A Simulation Tool for Dynamically Reconfiguable Field Programmable Gate Arrays. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems, Special issue on the 1995 IEEE ASIC conference*, 4(3):381–390, 1996. ISSN:1063-8210.

[125] Bjarne Stroustrup. *The C++ Programming Language (Special 3rd Edition)*. Addison-Wesley, 2000.

[126] Sun Microsystems, `http://java.sun.com/j2se/1.4/pdf/serial-spec.pdf`. *Java Object Serialization Specification*, rev.1.4.4 edition, Aug. 2001.

[127] Nozar Tabrizi, Nader Bagherzadeh, Amir H. Kamalizad, and Haito Du. MaRS: A Macro-pipelined Reconfigurable System. In *CF '04: Proceedings of the 1st conference on Computing frontiers*, pages 343–349. ACM, New York, NY, USA, Mar. 2004.

[128] Tensilica, Inc., `http://www.tensilica.com`. *Xtensa Architecture and Performance*, Sept. 2002.

[129] Kari Tiensyria, Yang Qu, Yan Zhang, Miroslav Cupák, Luc Rynders, Geert Vanmeerbeeck, Kostas Masselos, Konstantinos Potamianos, and Marko Pettisalo. SystemC and OCAPI-xl Based System-Level Design for Reconfigurable Systems-on-Chip. *Proceedings - Forum on Specification and Design Languages, FDL'04*, 2:428–439, 2004.

[130] T.J. Todman, G.A. Constantinides, S.J.E. Wilton, O. Mencer, Wayne Luk, and Peter Y. K. Cheung. Reconfigurable Computing: Architectures and Design Methods. *IEE proceedings. Computers and digital techniques*, 152(2):193–207, Mar. 2005.

[131] Steve Trimberger. Trusted design in FPGAs. In *Proceedings of the 44th annual conference on Design automation*, pages 5–8, New York, NY, USA, 2007. ACM.

[132] Vasutan Tunbunheng and Hidearu Amano. Black-Diamond: a Retargetable Compiler using Graphwith Configuration Bits for Dynamically Reconfigurable Architectures. In *SASIMI Proceedings (R4-12)*, pages 412–419, Oct. 2007.

[133] Milan Vasilko. DYNASTY: A Temporal Floorplanning Based CAD Framework for Dynamically Reconfigurable Logic Systems. *9th International Workshop on Field-Programmable Logic and Applications*, pages 124–133, Aug.-Sept. 1999.

[134] Milan Vuletić, Laura Pozzi, and Paolo Ienne. Seamless Hardware-Software Integration in Reconfigurable Computing Systems. *IEEE Design and Test of Computers*, pages 102–113, Sept.-Oct. 2005.

[135] Hasitha Muthumala Waidyasooriya, Weisheng Chong, Masanori Hariyama, and Michitaka Kameyama. Multi-context fpga using fine-grained interconnection blocks and its cad environment. *IEICE Transactions*, 91-C(4):517–525, 2008.

[136] Herbert Walder and Marco Platzner. Online Scheduling for Block-partitioned Reconfigurable Devices. *Proceedings of Design, Automation and Test in Europe (DATE)*, 2003.

[137] James Braxton Webb. Methods for Securing the Integrity of FPGA Configurations. Master's thesis, Bradley Department of Electrical and Computer Engineering, Blacksburg, Virginia, Aug. 2005.

[138] Michael J. Wirthlin. *Improving Functional Density Through Run-Time Circuit Reconfiguration*. PhD thesis, Brigham Young University, Nov. 1997. PhD thesis.

[139] Michael J. Wirthlin. Integrating the JHDL Design Environment into Ptolemy II. Ptolemy Miniconference, Mar. 2001. `http://www.ptolemy.eecs.berkeley.edu/conferences/01`.

[140] Michael J. Wirthlin and Brad L. Hutchings. DISC: The dynamic instruction set computer. In *Field Programmable Gate Arrays for Fast Board Development and Reconfigurable Computing*, pages 92–103, 1995.

[141] Xilinx, Inc., `http://www.xilinx.com/products/virtex4/index.htm`. *Virtex-4 Family Overview*, v3.0 edition, Sept. 2007.

[142] Xilinx, Inc., `http://www.xilinx.com/support/documentation/data_sheets/ds031.pdf`. *Virtex-II Platform FPGAs: Complete Data Sheet*, v3.5 edition, Nov. 2007.

[143] Xilinx, Inc. *Early Access Partial Reconfiguration User Guide*, ug208 (v1.2) edition, Sept. 2008.

[144] Xilinx, Inc. *Early Access Partial Reconfiguration User Guide (UG208)*, v1.2 edition, Sept. 2008.

[145] Xilinx, Inc., `http://www.xilinx.com`. *Spartan-3 FPGA Family Data Sheet*, ds099-4 (v2.4) edition, June 2008.

[146] Xilinx, Inc., `http://www.xilinx.com/products/virtex5/index.htm`. *Virtex-5 Family Overview*, v4.4 edition, Sept. 2008.