This Publication's Reference:

[1] Andreas Raabe and Armin Felke. A SystemC Language Extension for High-Level Reconfiguration Modelling. In *Forum on Design Languages and Specification (FDL'08)*, 2008.

A SYSTEMC Language Extension for High-Level Reconfiguration Modelling

Andreas Raabe Armin Felke
University of Bonn
Technical Computer Science
Römerstr. 164, 53117 Bonn, Germany
{raabe,felke}@cs.uni-bonn.de

Abstract

The ongoing trend towards development of parallel software and the increased flexibility of state-of-the-art programmable logic devices are currently converging in the field of reconfigurable hardware. On the other hand there is the traditional hardware market, with its increasingly short development cycles, which is mainly driven by high-level prototyping of products. This paper presents a library for modelling reconfiguration in the leading high-level system description language SYSTEMC *combining IP reuse and high-level modelling with reconfiguration. Details on the underlying simulation engine are given, which allows safe disabling and re-enabling of all process types without altering the kernel. Novel control statements and internal techniques that allow safe usage of process controlling in conjunction with standard* SYSTEMC *language constructs are presented. A real world case study using the presented library proves its applicability.*

1. Introduction

Due to increasing micro-miniaturisation in chip production hardware development evolved from plain circuit design into the development of complex heterogeneous systems with an increasing number of increasingly complex processing elements [11]. Not only that productivity of hardware designers does not grow as fast as the number of available transistors per chip (productivity gap), but the time to bring products to market is reduced as well. To fill the gap one obvious approach is reuse of own and externally produced components (IP-cores). The latter are usually provided closed source and remain intellectual property of the vendor. IP reuse is widely regarded as one of the major motors of productivity in contemporary chip-design.

To cope with complexity higher levels of abstraction were introduced in system design and simulation. They enable early estimation of time and hardware consumption. Especially the introduction of Transaction Level Design to evaluate the impact of bus models has proven to be highly efficient and productive in practice [7]. A large number of system description languages, mostly based on C/C++, have been proposed since [3]. Among them SYSTEMC became the most prominent one.

Configurable logic devices evolved considerably as well. State-of-the-art devices provide tremendous processing power and dynamic reconfiguration abilities, qualifying them as highly parallel co-processing units. This led to increased research in run-time reconfigurable systems, which are now close to commercial breakthrough [12].

To enable the design community to conveniently develop reconfigurable architectures in a short time-to-market, this paper presents the library RECHANNEL, which extends SYSTEMC with advanced language constructs for high-level reconfiguration modelling. Special focus lies on functional and transactional levels of modelling. Details of the underlying simulation library are given, which allows safe disabling and re-enabling of processes. The standard OSCI SYSTEMC-kernel is used without any changes.

2. Related Work

OSSS+R [10] is certainly one of the most prominent approaches towards high-level design methodology for reconfigurable hardware. It is based on OSSS [4], which extends the synthesizable subset of SYSTEMC by adding constructs that enable modelling, simulation and synthesis of object oriented features. The basic assumption of OSSS+R is, that reconfiguration can be interpreted as an exchange of two objects sharing a common base type and can therefore be modelled with polymorphism. Due to the fundamental change of the programming paradigm, from component based hardware design to object orientation, reuse of existing SYSTEMC IP-cores is impossible. Additionally, only passive objects are featured as reconfigurable components.

An approach towards a SYSTEMC reconfiguration extension closer to the original methodology was developed in the ADRIATIC project [13]. It targets transaction level (TL) description and simulation of reconfiguration aspects, focusing on early evaluation of the performance impact of reconfiguration. Therefore an automated tool is introduced that analyses static TL models and identifies components that could be made reconfigurable. The designer's task is now reduced to exploring the design space with respect to reconfiguration. The main limitation of the ADRIATIC approach is the restriction of the underlying interfaces. The target architecture needs a generic bus layout that fits into the supported interface scheme.

In [1] a modified `SystemC` kernel that features advanced process control statements is presented. In [2] a modified `SystemC` kernel is presented that allows enabling and disabling of processes. It is even used for modelling and simulation of a reconfigurable system. Both are altering the kernel and hence are not standard compliant.

To provide full coverage of SYSTEMC 2.2 and to enable some of the more advanced features presented in this article the RECHANNEL library was completely reimplemented. Since the user interface changed, a brief recap and update on the basic features presented in [8] is given in the next Section. Afterwards, Sec. 4 and Sec. 5 present novel language constructs and features.

3. Basic Reconfiguration Modelling

RECHANNEL was designed with respect to a number of objectives. Any SYSTEMC extension should *comply with the language standard IEEE 1666 [5]. IP-reuse should be possible* to enable designers to exploit the vast number of commercially available components. The library should *melt as seamlessly as possible with the* SYSTEMC *methodology* and *feature reconfiguration on all levels of abstraction* without a change in programming paradigm. Hand crafted refinement from functional to register transfer level should still be possible to allow *maximum efficiency in resource utilisation* and to *avoid imposing any design limitations*.

3.1. Interpreting Reconfiguration as Circuit Switch

Hardware designers often interpret reconfiguration as switching between two or more modules [6]. This very common approach is usually modelled using a bus that is either described as a module or a channel. The modules to be exchanged are then connected to the bus and can now be addressed only one at a time. The reconfiguration controlling can thus be described as an arbiter.

This approach has three main limitations: the tremendous development effort, delays caused by reconfiguration are usually not respected and the system's topology as well as its timing are altered.

RECHANNEL uses a technique that resembles reconfiguration buses, but does not have these grave limitations. *Portals* are introduced to connect the original channel to

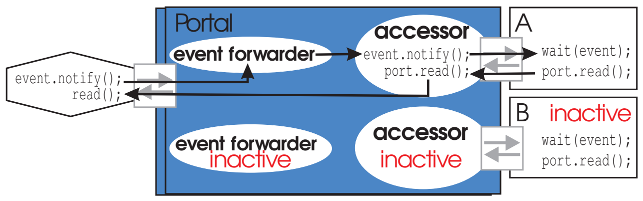

Figure 1: A portal connecting two reconfigurable modules to a standard SYSTEMC channel. A simulation sequence is shown, where a channel event is triggered by some outside source and is then forwarded to the accessor associated with the currently active module. A channel access within this module is triggered and is being executed via the accessor.

the reconfigurable modules' port[1]. The portal's function is twofold: Firstly, accesses of the active module to the channel need to be executed, while inactive modules should not be allowed to access the channel. Additionally, it is necessary to provide the module's port with an interface it is able to bind. This is both done by binding a so-called accessor object, which is part of the portal, to the port. It needs to be derived from the interface the port can connect to and forward interface accesses to the channel. Secondly, channel events need to be passed to the currently active module. Since the portal's accessors implement the interface the modules' port needs, they also possess the events provided by the interface. These events are now registered with event forwarders inside the portal. These components listen to the channel's events and notify the according events inside the accessor associated with the currently active module. Figure 1 illustrates this.

As will be described in Sec. 3.3, a portal's state is controlled by the modules connected to it. On the other hand the portals' state influences the modules' state, since only one reconfigurable module per portal is allowed to be active at a time.

The portal's type depends on the interface of the channel that it is bound to. Portals for all SYSTEMC channel interfaces are provided by the RECHANNEL library along with a toolkit that allows easy construction of portals for custom-built channels. This process could be automated as well, since it does not demand any creative coding, but is merely a repetition of information known to the compiler, but not available to the designer via C++ language constructs.

3.2. Creating Reconfigurable Modules from Static Ones

To allow IP reuse as demanded in Sec. 3, it is necessary to provide a mechanism to equip standard SYSTEMC modules with additional reconfiguration related features (i.e., configuration timings, bitfile size, etc.). In the RECHANNEL environment they also have to be able to efficiently interact with the portals they are connected to. Both can be achieved by deriving from the original static module and

[1]Hardware designers familiar with the Xilinx Virtex modular design flow, may think of portals as ”bus macros”.

rc_reconfigurable, which is a base class providing the necessary capabilities. To provide some of the more advanced mechanisms that will be described in Sec. 5 it is of some benefit to additionally wrap the original type into a template, which also automates this derivation. A reconfigurable version M_rc of a static module type M can now be derived.

```
class M_rc:
    public rc_reconfigurable_module<M>
```

The resulting type M_rc is also of type M and rc_reconfigurable. To provide more convenience ReChannel automates this process by providing a macro RC_RECONFIGURABLE_MODULE_DERIVED, which accepts the static module type as parameter and a macro RC_RECONFIGURABLE_CTOR_DERIVED, where the user can define reconfiguration related properties (e.g. the module's loading delay).

```
RC_RECONFIGURABLE_MODULE_DERIVED(M_rc, M) {
 RC_RECONFIGURABLE_CTOR_DERIVED(M_rc, M) {
   rc_set_delay(RC_LOAD, sc_time(1, SC_MS));
}}
```

3.3. Control

To control reconfiguration it would be tedious for designers to switch all portals manually. Hence a reconfigurable module can be requested to activate itself. This request is passed down to the portals, which allow switching only if no other module is registered as active. A module is only activated if all its portals can be switched. But this implicit control of the portal's state via rc_module is still not convenient enough. Therefore a simulation control object rc_control provides registration and reconfiguration control functions for modules. E.g. instantiating a control object, registration of three reconfigurable modules and activation of one of them via rc_control looks like this:

```
rc_control ctrl;
ctrl.add(m1 + m2 + m3);
ctrl.activate(m1);
```

Figure 2 shows an example of a reconfigurable design with two alternatively present modules. Their reconfiguration state is controlled by a custom configuration controller using rc_control.

Additionally, in more complex designs it might be necessary to simulate usage of multiple reconfigurable platforms with different reconfiguration behaviour. By deriving from rc_control and overloading a callback function called takes_time a simulation control object can be implemented that calculates reconfiguration timings from module attributes (i.e., bitfile size). This way properties of the reconfigurable hardware used for the design can be modelled

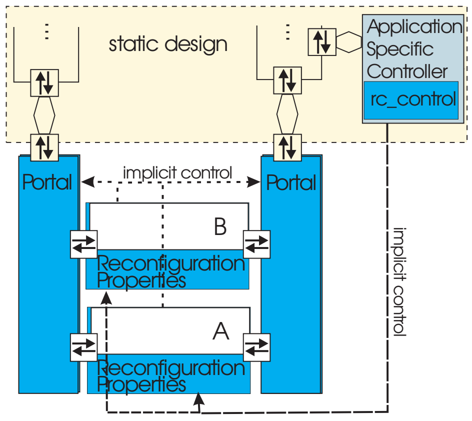
static design
Application Specific Controller
rc_control
Portal
implicit control
Portal
B
Reconfiguration Properties
A
Reconfiguration Properties
implicit control

Figure 2: Design with two alternatively present modules.

or different alternatives can be evaluated with respect to the impact on system performance.

4. Advanced ReChannel features

4.1. Exportals

A portal is a specially designed component to connect a channel of a design's static part to ports of reconfigurable modules. To allow reconfigurable modules to use exports as well as ports, the portal concept needs to be generalised. Therefore, its control interface was encapsulated into the rc_switch base class. rc_portal and the novel rc_exportal are derived from it. This way implicit control from module to portal and from module to exportal is implemented using the same mechanism. Additionally, an exportal needs to forward channel events in the opposite direction than a portal does, from reconfigurable to static end. Nevertheless, the same techniques that are implemented in a portal can be used. Interface accesses on the other hand are more difficult. Since they need to be forwarded from static parts of the design to reconfigurable ones, it can occur that *no* reconfigurable module is currently active to answer the request. Here two cases can be distinguished: If the access is blocking, the exportal can simply wait until a module is activated that can execute the request. But in case of a non-blocking access it must be executed immediately. If this occurs the design will in general be erroneous, but still the access has to be executed on some interface to allow the simulation to continue and issue a warning[2]. This is done by supplying a fallback channel that reacts accordingly.

4.2. Synchronisation

But blocking accesses can cause problems as well. Since portals are plugged between port and channel they can only gain control if an access is initiated, when it is finished or if a channel event is notified. In timed environments this does not cause any problems. But for untimed (and timed-

[2]Note that in case of resolved signals some other options are available as well, which are left out here.

untimed hybrid) modules it makes it difficult for the designer to control when the module is to be deactivated without input and output data becoming asynchronous. E.g. a module reading from an input port A and writing to an output port B must in general not be deactivated if it has read the input, but no output was written yet. Therefore it must be possible to either define blocks of code within the module's processes as atomic transactions, or to externally define synchronisation conditions depending on the module's communication behaviour. The former is the far more elegant solution and will be described in Sec. 5.1. Still, it rules out IP reuse and hence the latter approach is supported by ReChannel as well. Therefore synchronisation filters are provided, which allow bookkeeping of channel accesses by manipulating transaction counters. Only if all counters equal zero the module can be deactivated. Reconfigurable modules may now equip portals with these filters and define synchronisation conditions using the information supplied by the filters. E.g. let M be an IP core with the behaviour described above, then reading from input port A should increase and writing to output port B should decrease a transaction counter.

```
RC_RECONFIGURABLE_MODULE_DERIVED(M_rc, M) {
 RC_RECONFIGURABLE_CTOR_DERIVED(M_rc, M)
  filterA(tc,1),     // if data is read begin transaction
  filterB(tc,-1) {  // if data is written end transaction
    set_interface_filter(A, &filterA);
    set_interface_filter(B, &filterB);
 }
 rc_transaction_counter tc; // initially equals 0
 rc_fifo_in_filter<int>  filterA;
 rc_fifo_out_filter<int> filterB;
}
```

5. Explicit Description of Reconfiguration

ReChannel also comes with a set of language extensions intended for explicit description of reconfiguration. This is preferentially applied if a reconfigurable module is build from scratch, or if it is augmented with additional dynamic behaviour.

In constrast to native SystemC, ReChannel language constructs possess an implicit reset mechanism being triggered on reconfiguration. These language constructs are primarily comprised of classes, functions and macros corresponding to a particular functionality already known from SystemC (e.g. `rc_signal`, `rc_fifo`, `rc_event`, `rc_semaphore`, etc). With the availability of resettable components and processes, both structure and behaviour of reconfigurable modules can be modelled in an intuitive way without the need to care about additional logic that deals with reconfigurable behaviour itself.

Resetting a module accounts to resetting its processes and its sub-components (variables, channels and sub-modules). Hence all of the sub-components and contained

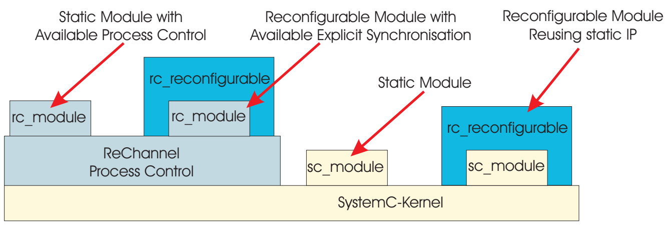

Figure 3: ReChannel's process control is layered on top of the SystemC infrastructure.

processes depend on a particular reconfigurable module up the SystemC object hierarchy tree, i.e., the first object among a component's parent list which is derived from class `rc_reconfigurable`. This object is denoted as *context module* of its sub-component. Whereas if no such parent exists, a component is said to be used in a *non-reconfigurable context*. As a general rule, resettable components and processes are optimised for utilisation within context modules. But they may also be employed in a non-reconfigurable context. All components provided by ReChannel are already equipped with the ability to implicitly reset themselves. A designer can use ReChannel's predefined components without further knowledge of the underlying mechanism. How to create a custom resettable component will be outlined in Sec. 5.2.

To enable process reset, ReChannel provides its own process registry and control API for internal management of resettable processes. The API directly builds upon standard SystemC functionality and therefore can be seen as an additional layer on top of SystemC's process infrastructure. Hence it does not need to alter the SystemC kernel and is therefore compliant to the IEEE 1666 standard [5].

5.1 Resettable Processes

In order to enable process reset, a process control layer is introduced (Figure 3). It integrates with the SystemC infrastructure by registering itself with SystemC's process registry and thus it is not necessary to alter the kernel implementation. Any process that is now registered with this process control layer instead of SystemC's native infrastructure can then be disabled, (re-)enabled and reset. Processes registered with SystemC will still remain fully functional and will not suffer from any performance loss. Processes with process control do not differ from standard SystemC processes and can thus coexist and interoperate with these even within a single module.

Macros available for process declaration are `RC_METHOD`, `RC_THREAD` and `RC_CTHREAD`. They correspond semantically to the respective SystemC process types, but are registered with ReChannel's process control layer.

Primary reset condition is the deactivation of the context module. Additional reset conditions may be assigned by the user when declaring the process. The invocation of `reset_signal_is()` will result in the process being reset on the occurrence of an edge of a signal. The reset behaviour of a process may be either set to be asynchronous (default for `RC_THREAD`) or synchronous (default for `RC_CTHREAD`).

```
// synchronously resettable thread process
RC_THREAD(proc); sensitive << clk.pos();
reset_signal_is(reset); rc_set_sync_reset();
```

Classes of type `rc_reconfigurable_module` and `rc_prim_channel`, amongst others, possess overloaded `wait()` and `next_trigger()` methods which are specially prepared for checking the reset conditions of a process. Consider the following example of a process function inside such a module.

```
void proc() {
        [...] // do something
  while(true) {
    wait(new_input_available);
        [...] // do something
} }
```

E.g. if the module is blocked by the `wait` and a reset is triggered due to deactivation of the context module, the execution of the process is cancelled. It will be restarted as soon as the context module is activated again.

The reset of thread processes is implemented using the C++ exception handling mechanism. Therefore, exception safety plays a major role in simulation reliability. Additionally, it is required that the executed code can be cancelled safely in respect of algorithmic correctness and data consistency. Cancellation of transactions or blocking operations, not specially designed to be cancellable, would render a design highly unreliable with respect to simulation stability and correctness. This implies that a reset mechanism requires fine-grained control of where and when a process may be reset. For this purpose ReChannel provides the macro `RC_NO_RESET` by which all reset signals can be temporarily disabled for the current process. The macro `RC_TRANSACTION` enables the designer to enclose blocks of code that must be finished before the reconfigurable context can deactivate.

```
x = input_fifo.read();    // read input (blocking)
RC_TRANSACTION {  // after data has been read
  y = calc(x);       // calculation must not be interrupted
  output_fifo.write(y); // write output
}                         // point of deactivation (if requested)
```

If a resettable process calls external code that is not intended for being cancellable at any time, a technical restriction is beneficial in this regard: For the reset of thread processes to work, it is required that these processes have previously been suspended in one of ReChannel's prepared `wait()` methods. Whereas if a thread process calls a function or interface method, that uses SystemC's native `wait()` functionality, the reset mechanism will be temporarily unavailable. Due to this characteristic a reset condition can be considered to be locally bound, e.g. within the borders of a module.

ReChannel also supports resettable spawned thread processes. In constrast to non-spawned processes these are considered to be temporary, i.e., they will be physically terminated if their context module is deactivated.

5.2 Resettable Components

Resettable components have the property that they can be automatically reset by ReChannel in case of activation or deactivation of their context module. ReChannel already provides resettable versions of all basic SystemC channels, (e.g. `rc_fifo`, `rc_signal`, `rc_signal_rv`,`rc_semaphore`, etc.) and the event class (`rc_event`). Additionally, the macro `rc_var()` allows declaration of resettable variables of arbitrary type.

```
rc_var(int, i); // declaration of resettable variable i
    [...]
i = 0; // initialisation of i within the constructor
```

A user-defined component can be easily made resettable by deriving it from `rc_resettable` and implementing its abstract base interface.

The particular state such a component is reset to can be assigned beforehand during the construction phase. At start of simulation the callback method `rc_on_init_resettable()` is invoked once on all resettables to give them opportunity to store their initial state after construction has finished. The request for an immediate reset is propagated by a call to `rc_on_reset()`.

```
class myComponent : public rc_resettable
{   [...] // implementation of myComponent
          // preservation of initial state
  virtual void rc_on_init_resettable()
    { p_reset_value = p_curr_value; }
          // definition of reset functionality
  virtual void rc_on_reset()
    { p_curr_value = p_reset_value; }
  int p_curr_value, p_reset_value;
};
```

For the reset mechanism to work, resettable components automatically register themselves with the current context module during construction. Hence the designer does not have to care about any further details.

6. Case Study

The ReChannel library was tested within the CollisionChip [9] project. Figure 4 shows the overall hardware/software project implementing a hierarchical collision detection with reconfigurable primitive test. It tests two objects for

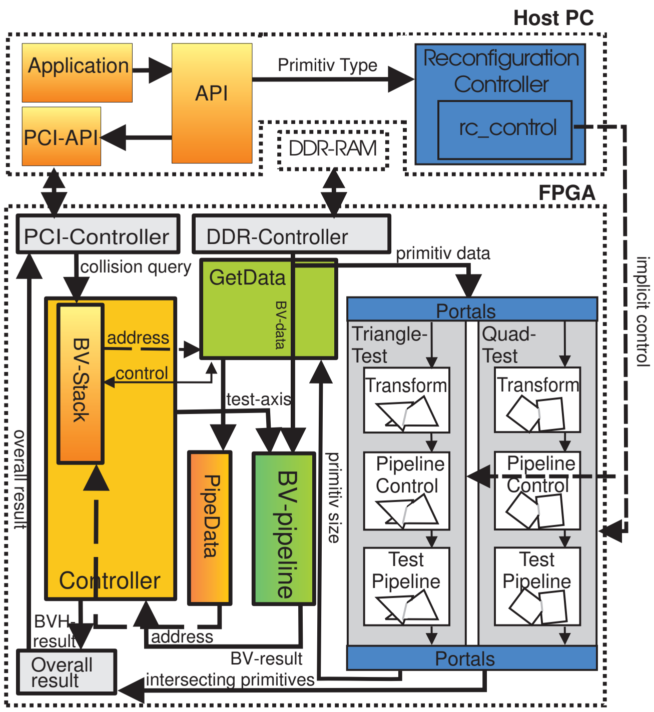

Figure 4: Hierarchical collision detection architecture with reconfigurable primitive test.

intersection. Depending on the primitive type the objects are constructed of, a primitive test is loaded into the FPGA. The main design preselects pairs of primitives that are very close to each other and hence might intersect. Already selected pairs are checked for intersection by the currently active primitive test.

Parts to be realized in hardware were implemented and tested on timed functional as well as on RT-level. Overall the RTL project consists of over 17000 lines of SystemC code. Introducing reconfiguration into simulation using ReChannel took a single developer only 2 days. The reconfigured modules were not altered, but treated as closed source components.

7. Conclusion and future work

This article presented a library for modelling reconfiguration in SystemC, which combines IP reuse and high-level modelling with reconfiguration. Advanced synchronisation techniques for high-level reconfiguration modelling were presented along with an internal process control. The latter allows safe usage within the SystemC framework by providing necessary synchronisation statements. It does not alter the SystemC kernel and complies to the language standard [5].

To cover all SystemC language constructs and to provide all of the above a full reimplementation of the library was provided and the basic mechanisms were presented. Some extended techniques (e.g. resolution of driver conflicts, make-up of variable reset functionality, etc.)were left out due to space restrictions. A case study was presented proving ReChannel to be effective and productive.

We are currently working on support for modelling mobility using ReChannel primitives, and providing a synthesis case study. Our next steps will be the release of ReChannel as an open source library and providing a discussion of functionality that should be incorporated into the SystemC standard to ease implementation of language extension libraries. We are also planning on providing portals and other components necessary to cover the TLM library.

References

[1] B. Bhattacharyya, J. Rose, and S. Swan. Language Extensions to SystemC: Process Control Constructs. In *DAC '07: Proceedings of the 44th annual conference on Design automation*, pages 35–38, NY, USA, 2007. ACM Press.

[2] A. V. Brito, M. Kuhnle, M. Hubner, J. Becker, and E. U. K. Melcher. Modelling and Simulation of Dynamic and Partially Reconfigurable Systems using Systemc. *ISVLSI*, 00:35–40, 2007.

[3] S. A. Edwards. The Challenges of Hardware Synthesis from C-Like Languages. In *DATE '05: Proceedings of the Conference on Design, Automation and Test in Europe*, pages 66–67, Washington, DC, USA, 2005. IEEE Computer Society.

[4] E. Grimpe and F. Oppenheimer. Aspects of Object Oriented Hardware Modelling With SystemC-Plus. In *System on Chip Design Languages. Extended papers: Best of FDL'01 and HDLCon'01*. Kluwer Academic Publ., 2002.

[5] IEEE Standards Association Standards Board. *IEEE Std 1666 -2005 Open SystemC Language Reference Manual*.

[6] P. Lysaght and J. Stockwood. A Simulation Tool for Dynamically Reconfigurable Field Programmable Gate Arrays. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 4(3):381–390, 1996.

[7] S. Pasricha, N. Dutt, and M. Ben-Romdhane. Extending the transaction level modeling approach for fast communication architecture exploration. In *DAC '04: Proceedings of the 41st annual conference on Design automation*, pages 113–118, New York, NY, USA, 2004. ACM Press.

[8] A. Raabe, P. A. Hartmann, and J. K. Anlauf. Rechannel: Describing and Simulating Reconfigurable Hardware in SystemC. *ACM Transactions on Design Automation of Electronic Systems*, 13(1):1–18, 2008.

[9] A. Raabe, S. Hochgürtel, G. Zachmann, and J. K. Anlauf. Space-Efficient FPGA-Accelerated Collision Detection for Virtual Prototyping. In *Design Automation and Test (DATE)*, pages 206–211, Munich, Germany, 2006.

[10] A. Schallenberg, F. Oppenheimer, and W. Nebel. Designing for dynamic partially reconfigurable FPGAs with SystemC and OSSS. In *Forum on Specification and Design Languages*, Lille, France, Sept. 2004.

[11] Y. Tanurhan. Processors and FPGAs Quo Vadis? *IEEE Computer*, 39(11):108–110, 2006.

[12] N. Tredennick and B. Shimamoto. The Rise of Reconfigurable Systems. In *Engineering of Reconfigurable Systems and Algorithms*, 2003.

[13] N. S. Voros and K. Masselos. *System Level Design of Reconfigurable Systems-on-Chip*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.