ReChannel: Describing and Simulating Reconfigurable Hardware in SystemC

ANDREAS RAABE, PHILIPP A. HARTMANN, JOACHIM K. ANLAUF
Technical Computer Science, University of Bonn, Germany

With the on-going integration of (dynamic) reconfiguration into current system models, new methodologies and tools are needed to help the designer during the development process. This article introduces a language extension for SystemC along with a design methodology for describing and simulating dynamically reconfigurable systems at all levels of abstraction. The presented library provides maximum freedom of description of reconfiguration behaviour and its control, while featuring simulation of run-time configuration, removal and exchange of custom modules as well as third-party IP-cores during the complete architecture refinement process. When designing at RT-level the resulting hardware description can easily be synthesized by standard synthesis tools.

Categories and Subject Descriptors: B.6.3 [**Logic Design**]: Design Aids—*Hardware description languages*; B.6.3 [**Logic Design**]: Design Aids—*Simulation*; B.7.2 [**Integrated Circuits**]: Design Aids—*Simulation*

General Terms: Design, Languages

Additional Key Words and Phrases: Reconfigurable hardware, dynamic reconfiguration, SystemC, refinement, simulation, hardware description

1. INTRODUCTION

To cope with the increasing complexity of recent system models, new methodologies and tools have been developed in recent years [Wolf 2003]. The modelling at higher abstraction levels at early design stages and the integration of external intellectual property (IP) is becoming more and more important with respect to time-to-market.

In addition to being a hot topic in research, (run-time) reconfigurable systems are close to their commercial breakthrough [Tredennick and Shimamoto 2003; Bouldin 2005]. Introducing reconfiguration properties into the system at a very early design stage is advisable. Since use of run-time reconfiguration often is prohibitively expensive, deciding if it will be integrated into a design needs to be done as early as possible in the design cycle. A variety of platforms and models have been proposed [Compton and Hauck 2002], but still tool support for modelling and synthesizing reconfigurable systems is somewhat limited, especially when it comes to application-specific solutions.

For instance, the de-facto standard for system modelling at higher abstraction levels, SystemC [Open SystemC Initiative (OSCI)], does not support changes to the system's module topology during simulation. This leads to difficulties in modelling of reconfigurable systems using this hardware description language (HDL). The RECHANNEL library,

Author's address: {A. Raabe, P. Hartmann, J. Anlauf}, Technical Computer Science, Römerstr. 164, 53117 Bonn, Germany, philipp.hartmann@offis.de,{raabe,anlauf}@cs.uni-bonn.de

which is introduced in this article, is an extension to SystemC, that overcomes these limitations, *without* actually changing the underlying simulation kernel.

To model the different system topologies, that can occur in a reconfigurable design, reconfigurable modules are "activated" and "deactivated" by conditionally intercepting the communication between static and reconfigurable parts of the design. This is achieved through the concept of *portals* (see Section 4.1). The use of *portals* allows the usage of any, even custom-built SystemC channel in a reconfigurable context, which leads to a highly flexible methodology for modelling reconfigurable systems.

Since the main modification of the system, that is to be extended with reconfigurability aspects, takes place within the interconnection of different parts of the system – i.e. between static and reconfigurable parts – no changes to existing modules are required. This allows inclusion even of third-party IP, where only the interface is available. Reconfiguration properties, like configuration times, can be added to those modules using inheritance.

The remainder of this article is organised as follows: The next section gives a more detailed insight into the objectives behind the presented approach while it is compared to other existing work in the field of reconfiguration within the SystemC context in Section 3. Section 4.1 introduces the *portal* concept and its applicability for custom channels is outlined. Creation of reconfigurable modules from static ones is described in Section 4.2. Afterwards, Section 5 shows an application of RECHANNEL within a hardware design on RT-level. It is demonstrated that using RECHANNEL does not impair simulation time. Section 6 introduces advanced RECHANNEL techniques for portal construction, puts RECHANNEL into context with the SystemC refinement strategy, and introduces the RECHANNEL technique for more accurate reconfiguration timing estimation. In Section 7 conclusions are drawn and planned directions of further work are outlined.

2. OBJECTIVES

The main objectives, that led to the development of the RECHANNEL library are outlined in this section. As already mentioned in the previous section, SystemC does not support modelling dynamic reconfiguration directly. This is due to the fact, that changes to neither the module hierarchy nor to the interconnection properties of a system are possible after the simulation has started.

Therefore, the main goal of the RECHANNEL library is to enable modelling and simulation of run-time reconfigurable systems – which obviously might *change* their hierarchy and/or interconnection characteristics during run-time – with SystemC. To achieve most flexible and powerful results, following important considerations were taken into account.

Independence of used SystemC kernel. Modelling and thus simulation of reconfigurable systems should not require usage of a specially crafted or manipulated SystemC simulation kernel. There are many commercially available tools (e.g. for co-simulation), that include their own SystemC run-time implementation. Excluding their use, due to the reliance on a non-standard simulation kernel, would be an unwanted limitation for the designer.

As a result, the RECHANNEL library has been designed to work with any SystemC simulator, that conforms to the *IEEE standard 1666 Open SystemC Language Reference Manual* [2005].

Re-using existing modules, IP and reconfigurability models. One of the major drawbacks of most approaches (see Section 3) to integrating run-time reconfiguration into hard-

ware descriptions is, that most of them require changes to modules, that should be "reconfigurable" within the system. Since reconfiguration is neither necessarily initiated nor controlled by such components itself, these changes are objectionable. Especially the usage of third-party IP-cores might be impossible within those frameworks. Even the required changes to existing (static) modules results in increasing development cost.

Integration into SystemC design flow. It is highly desirable to take possible (dynamic) reconfigurability schemes into account already at early stages of a system's design. Therefore, the possibility to study a system's behaviour in a reconfigurable context should be possible at all levels of abstraction within the SystemC design flow. Additionally, refinement of different modules (static *and* reconfigurable ones) should be possible independently of the inclusion and refinement of the reconfiguration properties (like scheduling techniques, development of a controller etc.) of the system. As a result, a language extension to SystemC, that allows modelling, simulation and refinement of a dynamically reconfigurable system, needs to support any, even custom-built channels natively.

To take full advantage of the SystemC refinement methodology control of the reconfiguration process should be as flexible as possible. Using RECHANNEL, the designer is free to model the controller as a module or even as a channel, so no limitation regarding the system's refinement is imposed. This is discussed in Section 4.3.

Synthesis. Still most systems are refined manually by a designer in order to provide maximum utilization of available resources. Even if automated synthesis of reconfigurable systems might lead to a shorter path to hardware, such an approach unavoidably imposes design limitations to the system. Hence it should be possible to refine and synthesize the system using standard techniques and tools independently of the reconfiguration properties.

3. RELATED WORK

Apart from specific description languages like JHDL [Bellows and Hutchings 1998], that support reconfigurable systems directly, or system-level approaches, that use UML (e.g. [Benkhermi et al. 2005]), several generic approaches [Schallenberg et al. 2004; Pelkonen et al. 2003; Alisson V. De Brito et al. 2006] have been proposed to model reconfigurable systems using SystemC. Additionally, SystemC, as a C++-based description language can be used at higher abstraction levels to develop systems which include reconfigurable parts. In this case, the modelling can be done using object-oriented techniques and avoiding the limitations of SystemC with respect to dynamic reconfiguration.

The ReConLib [Schallenberg et al. 2004] library integrates interchangeability of different behaviours using an object-oriented modelling approach. The underlying description language OSSS+R is an extension of OSSS [Grimpe and Oppenheimer 2002], which itself extends SystemC with synthesizable object-oriented features. The common interface of several exchangeable components is modelled as an inheritance hierarchy and different algorithmic behaviour is chosen through polymorphism. Specific reconfiguration properties (like partial state preservation, timing, etc.) can be expressed with OSSS+R as well.

Since this approach is based on an existing SystemC extension itself, the possibility of re-using existing SystemC modules is quite limited. The interface to the static parts of the system consists of OSSS+R specific procedure calls. Additionally, the reconfigurable components are not regular SystemC modules, but passive objects instead. As reasoned in the previous section, this usually requires the adoption of existing algorithms to the

OSSS+R methodology, and therefore the integration of external IP cores is not possible.

Within the ADRIATIC project (e.g. [Pelkonen et al. 2003; Tiensyrja et al. 2004]), the reconfigurable modules are modelled as bus slaves with strict requirements to the interface (read, write operations including an address). Several of these components are then combined to a so called *dynamically reconfigurable fabric* (DRCF). This DRCF component is used as a wrapper, that determines the required module and triggers the reconfiguration, if the addressed module is currently not "loaded" into such a DCRF component. Several use-cases have shown the usability of this approach (see [The Adriatic Consortium]).

The main limitation of the ADRIATIC approach is the restriction to the underlying interfaces. The target architecture needs a generic bus layout that fits into the supported interface scheme. Although this might not require a full reimplementation of existing modules, the limited communication model should not be necessary for a generic approach to design reconfigurable systems.

Another recently proposed SystemC extension [Alisson V. De Brito et al. 2006], is based on a modified simulation kernel. Basically, the modified kernel allows the explicit activation and deactivation of certain modules within a SystemC design. Since this approach might reduce the number of fired events during the simulation, it can be assumed, that a good simulation performance is achieved. On the other hand, as discussed in the previous section, need of a modified kernel is a major drawback. To the authors' knowledge no further detail (e.g. concerning simulation of reconfiguration timings) has been provided yet.

4. THE RECHANNEL APPROACH

4.1 Modelling reconfiguration on all levels of abstraction

Having a unified way of modelling reconfiguration at all levels of abstraction is highly desirable, not only because it is convenient, but since it simplifies data refinement as well as structural refinement. Additionally, it is necessary to enable the designer to leave most of the design untouched, when rendering parts of it reconfigurable. Especially inside the reconfigurable modules no changes should be necessary. Since otherwise integrating third-party IP, where only the interface is known, is simply impossible.

Nowadays hardware designers usually use buses to intercept communication between static modules to let them appear reconfigurable. This is an easy and most intuitive way to model that only one module out of a set of modules is currently able to communicate via a certain channel. In this simple scheme the channel's arbiter fills the task of a reconfiguration controller. This approach is named *dynamic circuit switching* and was proposed by [Lysaght and Stockwood 1996]. It comes in two different flavours: Firstly, the bus is modelled as a channel and substitutes the original channel. Secondly, it is made a module that connects to the original channel. Both are no completely satisfying solutions, since some drawbacks come with them in practice:

High development effort. For every SystemC channel type, that is used between static and reconfigurable modules, a custom "Reconfiguration Bus" (RecBus) has to be built from scratch, since the functional properties of the channel can differ considerably. In general the flexibility of such an approach will be very poor, if no extra effort is spent to allow connection of a random number of reconfigurable modules.

Reconfiguration cost. The dynamic reconfiguration of a system can usually not be performed instantaneously. The resulting delays might have an impact on the system's run-

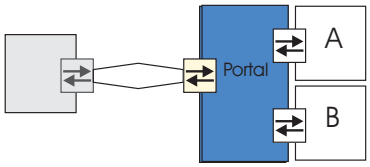


Fig. 1. Plugging a portal between a port and its channel allows interception of their communication. Binding multiple ports of different modules to a portal allows switching data between them.

time behaviour or even its functional correctness. Hence they have to be considered during development. With manually crafted RecBuses, the designer has to model the delays separately, which can be difficult and error prone.

Side effects. If modelled as a channel the RecBus needs to substitute the original channel. Hence, it needs to mimic the original channels behaviour, making it a full reimplementation. In addition to adding the switching capability, this makes it a time-consuming task even if simulation performance is ignored.

Describing the RecBus as a module connecting to the original channel unavoidably enforces that additional channels are used to connect the RecBus to the reconfigurable modules. This changes the system's topology and is error prone, since it is not necessarily clear which channel type is to be used or how it has to behave in case of reconfiguration. Furthermore, plugging another channel into the communication will in most cases change its timing behaviour, which might lead to unpredictable behaviour.

The next section introduces a way to intercept communication at the channel-to-module border, that resembles a RecBus, but does not have the limitations described above. Portals are introduced as a framework to facilitate construction of specialized switches between channels and modules that do not cause any changes in simulation timing (not even delta-cycles) by forwarding the channel's events and the reconfigurable modules' channel accesses on C++ language level.

The portals' state is controlled by the reconfigurable modules it is connected to. Reconfiguration delays, which are taken into account during simulation, can be modelled using `rc_modules` (see Section 4.2). The modules' state itself is controlled through a special simulation reconfiguration controller that can be used by the designer to model any custom controller and is presented in section 4.3.

4.1.1 *Using portals to intercept communication.* A *portal* is a special switch, designed to connect a static channel to a port of a reconfigurable module, see Figure 1.

During simulation, accesses to the corresponding port from within the reconfigurable module are then forwarded to the static channel. Additionally, any required events, the reconfigurable module is listening to (via sensitivity or dynamic `wait()` statements), is forwarded from the static channel to the module. Multiple reconfigurable modules can be bound to a single *portal* (Fig. 1). During simulation, the actual reconfiguration operations (see Section 4.3) change the data-flow through the portals depending on the reconfiguration state of the connected modules.

If all ports of a reconfigurable module are equipped with portals, no port can be triggered from outside, if the module is inactive (not configured). Therefore, no outbound traffic should occur any longer, since the module's processes are no longer triggered. Nevertheless, technically it is possible that a module keeps on triggering itself (for instance by

```
my_module_rc  mod;            // instantiate two reconfigurable modules
my_module2_rc mod2;

sc_fifo< int > fifo;          // instantiate (static) FiFo

rc_portal< sc_fifo_out<int> > portal;
                              // instantiate portal for sc_fifo_out<int> port

portal.static_port( fifo );   // connect the static channel
                              // to portal (named binding)

portal.bind( mod1.out );      // bind reconfigurable module's ports to portal
portal.bind( mod2.some_other_out );
```

Listing 1. Integrating a portal into a design is done analogously to the integration of a channel. Here the usage of a fifo portal is shown. The RECHANNEL library predefines portals for the standard SystemC ports.

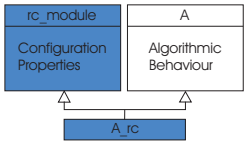


Fig. 2. Deriving from `rc_module` and a static module `A` results a reconfigurable module `A_rc`.

a member of type `sc_clock`). In this case outbound traffic is suppressed and a warning is reported to the designer.

For standard SystemC port types the corresponding portals are provided by the RECHANNEL library. Their usage is exemplarily shown in Listing 1. So the common RECHANNEL user will not need to construct portals himself.

Still, SystemC enables construction and usage of user-defined, possibly complex channels. Hence it is necessary to provide the user with an easy-to-use toolkit to devise portals for those custom channels. An approach that not only enables construction of portals for arbitrary channels but would also allow construction of a compiler that could do so is presented in Section 6.1.

4.2 Rendering own modules and third-party IP-cores reconfigurable

Still it is highly desirable to add reconfiguration features to the modules in question (i.e. reset and handshaking behaviour). Additionally, a library adding dynamic reconfiguration to the simulation without touching kernel code needs to add information and behaviour as well. In C++, like in most object-oriented languages, adding new abilities is usually done by inheritance. Hence the generic way to express, that a module `A_rc` is of type `A` and of type `rc_module` (reconfigurable module) is to derive it from both (see Fig. 2).

Since SystemC itself makes intensive use of inheritance this should be quite a familiar way of doing things for any SystemC user. Deriving `A_rc` from `rc_module` makes it a module that can be registered with a reconfiguration controller (see Section 6.2), and (with

```
class A_rc: public rc_module, public A {
protected:
 inline void rc_setup();
public:
 A_rc( sc_module_name name_ ): A(name_) {
   rc_init();        // Initialize reconfiguration
                     // behaviour of the module
   rc_setup();       // call rc_setup
 }
};
```

Listing 2. Example of a manual generation of a reconfigurable module. Module `A_rc` is derived from `rc_module` and static module `A`. The constructor starts the finite state machine that takes care of the reconfiguration state of the module and calls `rc_setup()` to reset the module at start-up.

```
inline void A_rc::rc_setup() {
  rc_reset<int>(i,0);                    // reset i to 0
  rc_preserve<sc_signal<int > > (j);     // preserve j

  set_loading_time(sc_time(20,SC_MS)); // module needs 20
                                         // milliseconds to load
  set_activation_time((sc_time(1.5,SC_MS));
                                         // and 1.5 milliseconds to activate
  set_removal_time((sc_time(2,SC_MS)); // preserving variables takes some time
}
```

Listing 3. Implementation of the `rc_setup()` method. In member function `rc_setup()` the integer member `i` is registered to be reset to zero and the member signal `j` to be preserved during reconfiguration. Modelling of configuration times is split into loading and activation delay.

```
RC_MODULE(A) {
  rc_reset<int>(i,0);                    // reset i to 0
  rc_preserve<sc_signal<int > > (j);     // preserve j
       //. . .
  set_removal_time(sc_time(2,SC_MS));    // preserving variables takes some time
}
```

Listing 4. A more convenient way of implementing a reconfigurable module using predefined macros.

some necessary preparations, see Section 4.1) be reconfigured.

Still, there are some things to cope with. A SystemC module will keep its state, i.e. its member variables will not be reset when removed and configured once again. A well known problem is, that hardware behaves differently. If not explicitly saved the modules

state is lost after reconfiguration. Therefore a special mechanism is necessary to obtain correctness of simulation. As proposed in [Schallenberg et al. 2004] a feasible solution to this is to demand explicit description of every variable's behaviour. Therefore the keywords `rc_preserve` and `rc_reset` are defined. Registering members to be preserved or reset is done in a special setup method `rc_setup()` (see Listing 2). If unspecified, a variables behaviour is undefined and the designer has to take care for correctness by other means, i.e. setting a reset signal of the module in question. In `rc_setup()` the simulated configuration and removal time of the module can be specified. Here we distinguish between the time to load the module into the FPGA's configuration memory and the time to actually activate it.

Last but not least, for the generation of a reconfigurable module it is necessary to call the function `rc_init()` that initializes all reconfiguration properties from within the module's constructor. To simplify this process the macro `RC_MODULE()` can be used alternatively. Listing 4 shows that a condensed and elegant description results.

4.3 Controlling Reconfiguration Simulation Control

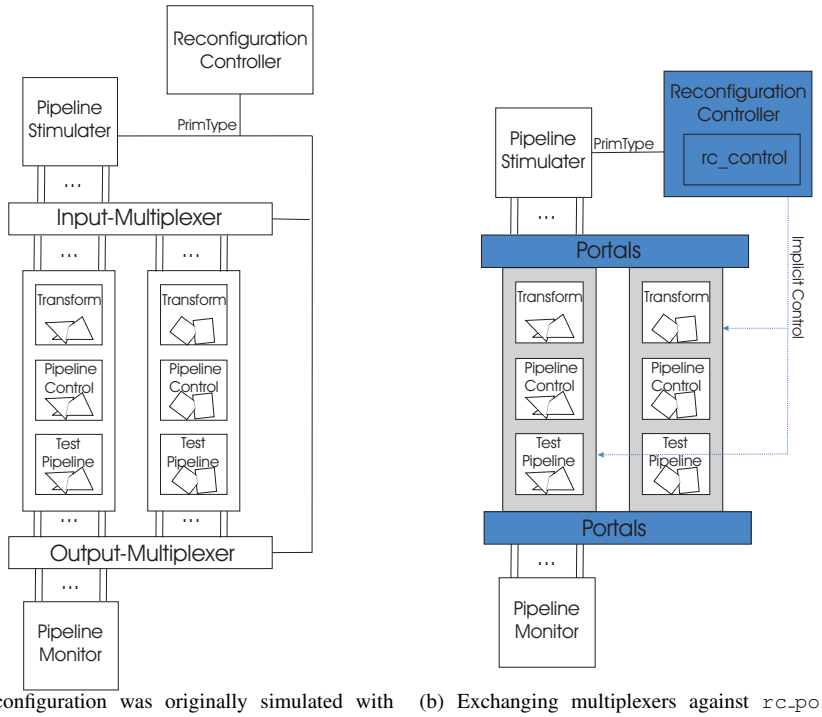
Since portals can be connected to ports of reconfigurable modules, their reconfiguration state depends on the states of the attached modules. The major advantage of this is that the designer does not need to take care for every portals state individually; manipulating module states suffices.

To take care of the simulation aspects of reconfiguration a control unit `rc_control` is necessary, that administrates the reconfigurable modules in the design and allows manipulation of their reconfiguration states.

```
rc_control ctrl;                          // create control object

ctrl.add(mod_1+mod_2+mod_3+mod_4); // register four modules with
                                          // reconfiguration simulation control
ctrl.add(mod_5);                          // register one more module

ctrl.load(mod_1);                         // load module 1

ctrl.activate(mod_1);                     // activate module 1

ctrl.activate(mod_2);                     // activate and load module 2
```

In order to enable the designer to model pre-fetching and variable preservation, loading and activation of modules are separated functions. If a module is to be activated but not loaded before, it is loaded automatically. Loading (or activating) a module takes at least the loading (or activation) time specified in the modules `rc_setup()` method (see section 4.2). Removal and deactivation functions can be used analogeously.

Activation and deactivation can consume even more time if the modules state can not be changed immediately. A module can not be activated for example if any of the portals still has another active module attached to it. This second module needs to be deactivated first. (There do exist some special cases where it is very valuable to have more than one active module per portal, hence this is supported as well.) On the other hand a module will not be deactivated as long as it has any blocking accesses pending on any of its ports. This mechanism is implemented in a state machine within `rc_module`. To enable the designer



(a) Reconfiguration was originally simulated with multiplexers. If the reconfiguration controller signals reconfiguration via the `primType` signal input and output multiplexers switch between primitive tests.

(b) Exchanging multiplexers against `rc_portals` simplifies the design. Portals connect to modules directly and control of the reconfiguration state is now done implicitly via RECHANNEL statements.

Fig. 3. Intersection test design with two modules that are interchanged at run-time. Triangle-triangle and quad-quad test consist of three submodules each.

to exploit or circumvent this functionality at his will RECHANNEL offers several reconfiguration state manipulation functions in `rc_control`, such as blocking (de-)activation, non-blocking (de-)activation and forced (de-)activation. Blocking functions wait until the according action is possible and return afterwards. Their non-blocking counterparts try if the state change can take place and report if it did or not. The forcing functions change the state without respect to any of the above mentioned constraints. Therefore they should be used with great care, since unpredicted effects may result. Still, since one of the objectives of RECHANNEL is to give maximum freedom to the designer these functions are provided since in some rare cases their usage might be inevitable when using third-party IP in a reconfigurable environment.

5. A PERFORMANCE STUDY

To investigate on applicability and performance of RECHANNEL it was integrated into a reconfigurable design. The simulated hardware is designed to test two graphical primitives for intersection. The kind of primitives to be tested can be changed to provide flexibility when used to accelerate physically-based simulations. It was developed within the COLLISIONCHIP project funded by the Deutsche Forschungsgemeinschaft (DFG) for high-speed collision detection using dedicated hardware. That makes it a real world example. Fully synthesizable pipelines for triangle-triangle and quad-quad tests are contained

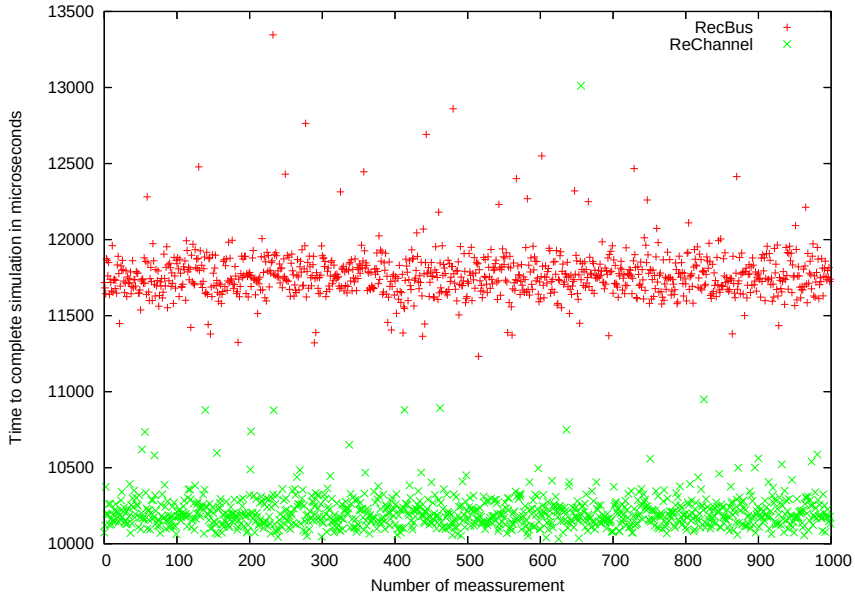


Fig. 4. 39610 pairs of triangles and quadrangles were checked for intersection interrupted by a reconfiguration. This was repeated 1000 times. Using RECHANNEL for simulation is approx. 15% faster than using multiplexers.

within the design and can be switched according to the kind of data provided by the stimulator. Originally the exchange of a triangle vs. triangle test against a quad vs. quad test during run-time of the simulation was simulated with multiplexers (see Fig. 3(a)), which can be interpreted as the simplest type of reconfiguration bus.

The design was deliberately chosen to be simple, to allow modeling without SystemC extensions. Otherwise a comparison of simulation run-time would not have been possible.

Exchanging the multiplexers with `rc_portals` simplifies the design (see Fig. 3(b)). As described in Section 4.1 portals connect to modules directly, all signal that where originally connecting the multiplexers to the pipelines are eliminated. A simple controller module encapsulates switching to the requested collision test. Control of the activation state is now done implicitly via RECHANNEL statements. No variables were reset to provide comparability of run-time measurements between ReChannel and the multiplexer solution. Modelling (re-)configuration times with RECHANNEL does not cause any extra cost in speed of simulation as long as the course of calculation done by the simulated design is not affected.[1]. But since this is an aspect of (re-)configuration that can not be grasped by the multiplexer solution at all, this would spoil the comparison. Furthermore, using RECHANNEL accuracy of the simulated timing behaviour is arbitrarily precise, depending on accuracy of the platform description in use, as will be described in section 6.3. Hence, respecting reconfiguration timings here would not result any extra information.

Comparing run-time of the simulation was done on a dual core Pentium D system at 3.20GHz with 1GB of main memory. 39610 pairs of triangles and quadrangles were checked for intersection interrupted by a single reconfiguration. As can be seen in fig. 4

[1]This is a rare case, but can occur e.g. if the sequence of a tree traversal depends on certain calculation's run-time.

this test was repeated 1000 times. Note that the design using RECHANNEL was simulating approximately 15% faster than the multiplexer solution. This is due to the overhead caused by the channels from multiplexers to modules, which are removed by using the RECHANNEL solution. Of course run-time of the simulation is strongly influenced by the time consumption of the algorithmic behaviour within the different modules and frequency of reconfiguration. But still this result shows that using RECHANNEL does not increase simulation overhead with respect to time, but can be expected to run even faster than a hand-crafted switch.

6. ADVANCED RECHANNEL FEATURES

6.1 Creating custom portals

As already stated in section 4.1 it is not possible to provide portals for custom build channels before they exist. Hence RECHANNEL provides the designer with an easy-to-use toolkit to devise portals for custom channels.

Basically, two steps have to be performed: Firstly, it is necessary to implement an `accessor` for the channel, that sets up the forwarding calls and declares the required event finders. Secondly, a so called `rc_port_traits` template specialisation is required, that specifies which interface/port/accessor type combination is needed for a given port. For both tasks, the RECHANNEL library provides helper macros, where applicable.

Forwarding channel accesses. For every channel, a so called `accessor` object has to be implemented. For the standard SystemC channels, these accessors are already part of the RECHANNEL library.

The purpose of these accessor objects is to allow forwarding of channel accesses and events. To enable forwarding of a channel access, *blocking* and *non-blocking* methods have to be distinguished. For both types of channel methods, the RECHANNEL library provides a macro, that takes care of the communication interception, if the calling module is currently inactive. As a result, the re-implementation is reduced to the choice of the correct macro and its usage around the "real" channel call. Listing 5 shows an example of the reimplementation of a blocking write function `bl_write`. Additionally, the *event finders*, i.e. the methods of the channel interface, that are used to access the channel's events, have to be re-implemented in the accessor as well. For convenience, the RECHANNEL library provides macros for this as well.

Forwarding channel events. If the accessor is implemented for all corresponding ports (either the generic `sc_port<my_interface>`, or a specific custom port e.g. `my_port`) of a given interface `my_interface`, the implementation of the corresponding portal `rc_portal<my_port>` is nearly ready.

All it takes, is the implementation of a template specialisation of some traits of the given port. `rc_port_traits` encapsulate the correct types of the interface/port/accessor combination and provide a static method, that specifies the events, that are to be forwarded. An example is shown in listing 6. As a result, the portal for the given port is ready to use. A slight limitation of this approach is the requirement of a fairly recent C++ compiler, that supports partial template specialisation.

This way introducing run-time reconfiguration is easy on all levels of abstraction featured by SystemC, especially on Transaction-Level where mainly custom build channels are used. Therefore the refinement process for custom build parts of the design can remain

```
template< class T >
class my_accessor :                          // inherit from rc_accessor template
  public rc_accessor< my_channel_if<T> > {
  RC_EVENT_FINDER( my_event );               // declare event finder
public:
  void bl_write( const T& data ) {           // enable forwarding of method
      RC_BLOCKING_ACCESS(
        this->channel->bl_write( data ));
  }
};
```

Listing 5. Create a custom accessor by implementing forwarding methods and declaring required event finders.

```
template< class T >
class rc_port_traits< my_port<T> > {
public:
  typedef my_channel_if<T> if_type;        // required type information
  typedef my_port<T>       port_type;
  typedef my_accessor<T>   accessor_type;

  static rc_event_list     events() {      // events to be forwarded
    return ( RC_EVENT( my_event ) );
  }
};
```

Listing 6. Specify interface, port and accessor types to create a new portal for a given port.

unchanged, with the single exception that reconfiguration needs to be taken into account (see section 6.2).

Additionally, this approach not only enables construction of portals for arbitrary channels but would also allow construction of a compiler that could do so. This is due to the fact that portal construction does not depend on any creative coding of the designer, but merely is a repetition of facts known to the compiler, but not available via C++ language constructs (e.g. the type of the interface passed to `sc_port` as template parameter).

6.2 Integrating reconfiguration into the refinement process

Introducing reconfiguration at a very early stage is advisable. Since use of run-time reconfiguration often is prohibitively expensive, deciding if it will be integrated into the design in question needs to be done as early as possible in the design cycle.

Functional Level. If done "by the book", a coarse approximation of the timing behaviour of a design is generated at timed-functional level. With RECHANNEL this can be done easily for reconfiguration timings. In the `rc_setup()` function the designer can set the time a reconfigurable module needs to be configured into or removed from the hardware (see Listing 3).

To control configuration on functional level it suffices to instantiate an object of type

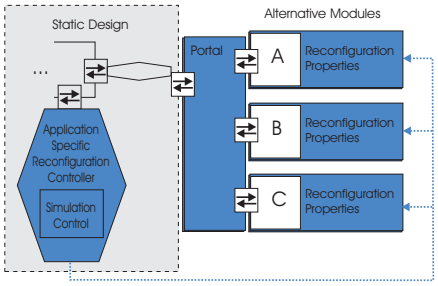


Fig. 5. An overview of a reconfigurable design on Transaction-Level. The reconfiguration controller is modelled as a hierarchical channel. The reconfigurable modules remain in their scope.

`rc_control` somewhere in the design and to register the reconfigurable modules with it as described in Section 4.3. Now the modules configuration state can be manipulated via function calls to `rc_control` whenever necessary. If it turns out, that reconfiguration requests need to be processed very fast or that most requests are coming from modules to be implemented in hardware it might be necessary to implement a hardware reconfiguration controller as well.

Since functional level modelling is mainly used to determine the module structure, a possible next step is encapsulating the reconfiguration controlling into a controller module instantiating `rc_control`. This way requests for a certain module are modelled more explicitly, since modules requesting use of a reconfigurable module need to inform the reconfiguration controller of this. Using RECHANNEL, the user is completely free to choose a scheduling strategy and communication interface for the controller.

Transactional Level. A slightly different approach is to use a hierarchical channel to encapsulate the controller (Fig. 5), when the refinement proceeds to Transaction-Level. Since requesting use of a reconfigurable module can be regarded as a request to a (sometimes very slow) bus, standard techniques and tools for the investigation on TLM timing behaviour can be used.

Different to other approaches, that implement the complete reconfigurable area or the reconfigurable modules as buses the reconfigurable modules remain in their original scope and the surrounding design has to be changed only very slightly. Obviously the topology of a static design remains basically unchanged if some parts are made reconfigurable. As discussed in Section 3, this is a major advantage.

Register Transfer Level / Synthesis. Evolving the dynamically reconfigurable design into a synthesizable hardware description requires the refinement of the reconfigurable modules as well as the refinement of the reconfiguration controller.

The different reconfigurable modules can be refined independently of the fact, that they are used in a reconfigurable environment, since their algorithmic behaviour is not directly affected by the reconfiguration. Standard SystemC synthesis tools can then be used to translate the modules e.g. to VHDL to allow further processing in the vendor-specific synthesis flow for dynamically reconfigurable systems. The only exception here is that the designer has to care for the preservation of internal register values during removal and reconfiguration, where needed.

Refining the controller to a synthesizable description can be done in two stages. Firstly,

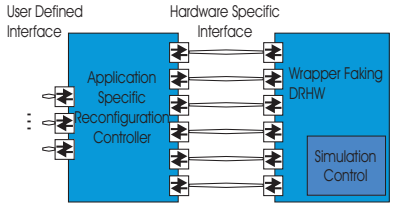


Fig. 6. The application specific part of the controller is refined to a pin-accurate model. Therefore a placeholder module for the reconfiguration behaviour of the underlying hardware is necessary. This can be build easily using the RECHANNEL simulation control `rc_control`.

the hierarchical channel is changed into a module with a pin-accurate interface by standard techniques (like adapter insertion and adapter inlining). Secondly, by encapsulating the properties of the dynamically reconfigurable FPGA (DRFPGA) in use into another module. The actual controller still contains the scheduling technique and the communication with the rest of the design. It can now be refined to synthesizability.

The physical reconfiguration interface on the target hardware can now be described in a behavioural fashion as a second module. It will not be synthesized but serves as a placeholder for the real DRFPGA and the hardware environment (DRHW) for simulation purposes. To mimic this behaviour `rc_control` can be instantiated and its activation and deactivation functions can be called from within the placeholder module (see Fig. 6). This placeholder module can be re-used for all designs using the same underlying DRHW.

The synthesis of the overall dynamically reconfigurable system might require specific, pin-accurate communication primitives (e.g. bus macros for Xilinx FPGAs) at the border of the static parts and the reconfigurable parts of the design. At these borders, the SystemC/RECHANNEL model contains the portals, which can be easily replaced by the appropriate communication primitives either manually or by a future preprocessing step during the SystemC synthesis.

6.3 More Accurate Modelling of Reconfiguration Timings

Modelling reconfiguration delays using estimates as it is described in Section 4.2 is an important first step in design space exploration. This way first information can be gained, if the model still meets its performance constraints when dynamic reconfiguration is used. But for a final decision in favour of reconfiguration, this will usually not suffice. More precise timing information will be needed to fine tune algorithms or to decide which hardware platform will be targeted. Therefore RECHANNEL offers a mechanism that allows description of specialized simulation controllers that mimic reconfiguration timing of a target platform.

Let `A_rc` be an `rc_module` and `PlatformProperty` to be a platform dependent property type. `PlatformProperty` can now contain additional module properties that depend on the target platform. (e.g. *bitfile size* if synthesized for the according platform). Now `A_rc` can be equipped with the platform's properties. (e.g. specify its *bitfile size*).

A specialized simulation controller `PlatformControl` can now be derived from `rc_control` that calculates reconfiguration timings based on these module properties. This can be done by overloading the `takes_time()` member function of `rc_control`.

A reconfigurable module can even be equipped with properties of multiple platforms and hence behaves differently under control of different controllers. This enables investigation

on the impact of different hardware platforms on the system's performance.

Using platform dependent properties accuracy of reconfiguration timings only depends on accuracy of the platform's behaviour and the property estimation. The latter will usually still be a guess. But estimating a circuit's size for instance, will usually be much more precise than directly guessing reconfiguration timings. Calculating reconfiguration timings out of properties will usually not be very difficult. For example, for a Xilinx Virtex-4 FPGA, the bitfile size has to be divided by the block size (1 or 4, depending on whether the internal configuration port (ICAP) is running in 32 Bit mode) and dividing the result by the clock frequency the ICAP is running on.

7. CONCLUSIONS AND FURTHER WORK

Accurate modelling of reconfigurable systems is not supported by SystemC natively, since the system's module topology can not be manipulated at run-time. To overcome this limitation, the RECHANNEL library was developed and presented in this article. It provides a powerful methodology for describing reconfigurable systems, even including third-party IP cores, and can be integrated into the refinement process most intuitively.

Furthermore, it neither requires any changes to the SystemC simulation kernel nor does it depend on any other library. RECHANNEL is build exclusively with SystemC language constructs conforming to the *IEEE standard 1666 Open SystemC Language Reference Manual* [2005]. Therefore it is independent of the SystemC simulator used. The only hard restriction is the necessity to use a compiler supporting partial template specialisation.

One major motivation to develop the presented methodology was the necessity for an applicable SystemC reconfiguration simulation environment within the COLLISIONCHIP project [Raabe et al. 2005;2006b;2006a]. Hence, applying the presented methodology to further real world problems in hardware accelerated collision detection will be one of our next steps. Exhaustive investigations on the simulation overhead caused by reconfiguration will be included as well.

To further ease usage of RECHANNEL automated generation of accessors and thus of portals will be included, as well as automated generation of reconfigurable modules from static ones. Therefore a syntactic analysis of the processed design is necessary, which is currently under development. Here, UNIWARE [Hartmann and Anlauf 2004] will be used as an intermediate representation. A major step towards complete coverage of all SystemC communication primitives will be the inclusion of exports into the framework.

Last but not least, the support for modelling mobility aspects – i.e. moving modules within the design – is considered for incorporation into the RECHANNEL framework as well.

ACKNOWLEDGMENTS

We would like to thank our students B. Bales, T. Becker, T. Loraing, M. Nolden, R. Reifenhäuser, U. Schuster, R. Thesen, and J. Wolf for their excellent work during the implementation of the RECHANNEL library.

REFERENCES

ALISSON V. DE BRITO, ELAMR U. K. MELCHER, AND WILSON ROSAS. 2006. An open-source tool for simulation of partially reconfigurable systems using SystemC. In *IEEE Computer Society Annual Symposium on Emerging VLSI Technologies and Architectures (ISVLSI'06)*. 434–435.

BELLOWS, P. AND HUTCHINGS, B. 1998. JHDL - An HDL for Reconfigurable Systems. In *FCCM, IEEE Symposium on FPGAs for Custom Computing Machines*. 175.

BENKHERMI, I., BENKHELIFA, A., CHILLET, D., PILLEMENT, S., PRÉVOTET, J.-C., AND VERDIER, F. 2005. System-Level Modelling for Reconfigurable SoCs. In *20th Conference on Design of Circuits and Integrated Systems (DCIS)*. Lisboa, Portugal.

BOULDIN, D. 2005. Enabling killer applications of reconfigurable systems: Ersa keynote and introduction. In *ERSA*, T. P. Plaks, Ed. CSREA Press, 7–16.

COMPTON, K. AND HAUCK, S. 2002. Reconfigurable Computing: A Survey of Systems and Software. *ACM Computing Surveys 34,* 2.

GRIMPE, E. AND OPPENHEIMER, F. 2002. 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., 213–223.

HARTMANN, P. A. AND ANLAUF, J. K. 2004. On Actors and Objects – OOP in System Level Design. In *FDL'04 – Forum on Specification and Design Languages*. Lille, France.

IEEE STANDARDS ASSOCIATION ("IEEE-SA") STANDARDS BOARD. 2005. *IEEE Std 1666 -2005 Open SystemC Language Reference Manual*.

LYSAGHT, P. AND STOCKWOOD, J. 1996. A Simulation Tool for Dynamically Reconfigurable Field Programmable Gate Arrays. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems 4,* 3, 381–390.

PELKONEN, A., MASSELOS, K., AND CUPAK, M. 2003. System-Level Modeling of Dynamically Reconfigurable Hardware with SystemC. *Proceedings of International Symposium on Parallel and Distributed Processing (Reconfigurable Architecturs Workshop)*.

RAABE, A., BARTYZEL, B., ANLAUF, J. K., AND ZACHMANN, G. 2005. Hardware Accelerated Collision Detection — An Architecture and Simulation Results. In *Design Automation and Test (DATE)*. IEEE Computer Society, Munich, Germany, 130–135.

RAABE, A., HOCHGÜRTEL, S., ZACHMANN, G., AND ANLAUF, J. K. 2006a. Hardware-Accelerated Collision Detection using Bounded-Error Fixed-Point Arithmetic. *Journal of WSCG '2006*, 17–24.

RAABE, A., HOCHGÜRTEL, S., ZACHMANN, G., AND ANLAUF, J. K. 2006b. Space-Efficient FPGA-Accelerated Collision Detection for Virtual Prototyping. In *Design Automation and Test (DATE)*. Munich, Germany, 206–211.

SCHALLENBERG, A., OPPENHEIMER, F., AND NEBEL, W. 2004. Designing for dynamic partially reconfigurable FPGAs with SystemC and OSSS. In *Forum on Specification and Design Languages*. Lille, France.

TIENSYRJA, K., QU, Y., ZHANG, Y., MIROSLAV, C., RYNDERS, L., VANMEERBEECK, G., MASSELOS, K., POTAMIANOS, K., AND PETTISALO, M. 2004. Systemc and ocapi-xl based system-level design for reconfigurable systems-on-chip. In *Forum on Design Languages (FDL)*.

TREDENNICK, N. AND SHIMAMOTO, B. 2003. The Rise of Reconfigurable Systems. In *Engineering of Reconfigurable Systems and Algorithms*, T. P. Plaks, Ed. CSREA Press, 3–12.

WOLF, W. 2003. A decade of hardware/software codesign. *Computer 36,* 4, 38–43.