A Methodology for Modelling and Simulation of Dynamic and Partially Reconfigurable Systems

Alisson Vasconcelos Brito[1], George Silveira[2] and Elmar Uwe Kurt Melcher[2]
[1]Federal University of Paraiba (UFPB),
[2]Federal University of Campina Grande (UFCG)
Brazil

1. Introduction

In the present day, partial reconfiguration is a reality (Becker & Hartenstein, 2003). There are many industries investing as well in fine-grain (like FPGAs (Huebner et al., 2004)) as in coarse grain solutions (eg. XPP (Becker & Vorbach, 2003)). This capability enables the necessary configuration area to decrease and the development of lower cost and more energy efficient systems, where timing is the main concern.
The main contribution of this work is to enable the engineers to discover earlier during the design-flow the best cost-benefit relationship between configuration time and saved chip area.
Such relationship is generally obtained only after the prototyping phase during the hardware verification. Once the dynamic reconfiguration simulation is possible in a simple way, the concrete benefits of such simulations can be checked in a simple way.
The innovative technique presented here allows the modeling and simulation of such systems by enabling new functions to module blocking and resuming in the simulator kernel. This enables the dynamic behavior to be foreseen before the synthesis on the target configuration (like FPGA). Furthermore, systems evaluation is possible even before their hardware description using a Hardware Description Language. Papers were published (Brito et al., 2006; Brito et al., 2007) presenting how the partial reconfiguration can be practically simulated.
In this work a novel methodology for simulate partial and dynamic reconfigurable system is presented. This methodology can be applied to any hardware simulator which uses an event scheduler. The main idea is to register each block that is not configured on a chip at a given moment in simulated time. Modifying the simulator scheduler, it is programmed to not execute those blocked modules. We prove in this work that this approach covers every partial and dynamic reconfigurable system situation. SystemC is used as a case of study and several systems were simulated using our methodology.
The section 2 presents what a simulator should implement to be considered able to simulate partial and dynamic systems. The methodology is presented on section 3 and section 4 presents how we applied it to SystemC. A particular strategy was adopted to log the chip area usage enabling the investigation of the benefits of dynamic reconfigurations in each application. This logging strategy is presented on section 5. Section 6 proves that the partial and dynamic reconfiguration can be really modeled and simulated using our methodology

Source: Dynamic Modelling, Book edited by: Alisson V. Brito,
ISBN 978-953-7619-68-8, pp. 290, January 2010, INTECH, Croatia, downloaded from SCIYO.COM

in practice with SystemC. Section 8 brings some consideration on the simulator performance after its adaptation and section 9 reports some further works using this methodology applying it to other targets.

2. Simulation of partial and dynamic reconfiguration

Before presenting the novel methodology for simulating partial and dynamic reconfiguration, it is necessary to characterize what in fact can be considered a simulator for dynamic reconfiguration. In (Lysaght & Dunlop, 1993) is described partial reconfiguration as the execution of a tasks sequence by hardware modules scheduled on time. In (Zhang & Ng, 2000) is affirmed that in order to simulate the operation of a Dynamically Reconfigurable FPGA (or DR-FPGA) a simulator must be able to simultaneously model any active static circuit and the switching of dynamic circuits along the time.
In (Dorairaj et al., 2005) is presented best practices for modelling partial reconfiguration using the PlanAhead simulation tool. It mainly recommends the utilization of bus macros among candidate modules for replacement. During the module substitution the original module is deactivated in order to activate the replacing one. The deactivation and activation of modules are the two basic operations for partial reconfiguration simulation.
Meanwhile, Pleis et. al. defend that a dynamically reconfigurable system is formed by different interchangeable functionalities (Pleis & Ogami, 2007).
Based on those interpretations of simulation of partial and dynamic, we can summarize that all simulators should be complete if it can model three operations:

- Module removing;
- Module switching;
- Module partitioning.

These basic operations are presented here. Fig. 1 presents a Module C being removed to give place to another module of same area of smaller.

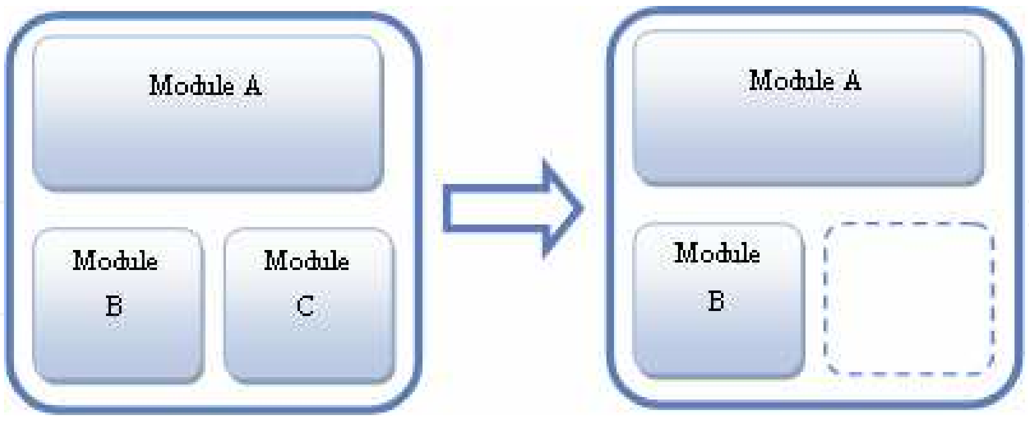


Fig. 1. Module removing of Module C.

Fig. 2 presents the second dynamic reconfiguration case, which can be seen as a logical continuation of module removing, when the Module C, after being removed, is replaced by a different module (Module D) on the same area.
Module partitioning is the third type of reconfiguration and is presented in Fig. 3. On this illustration the Module A is separated into three different modules, which together execute the same functionality of Module A, but by separated modules at different moments.

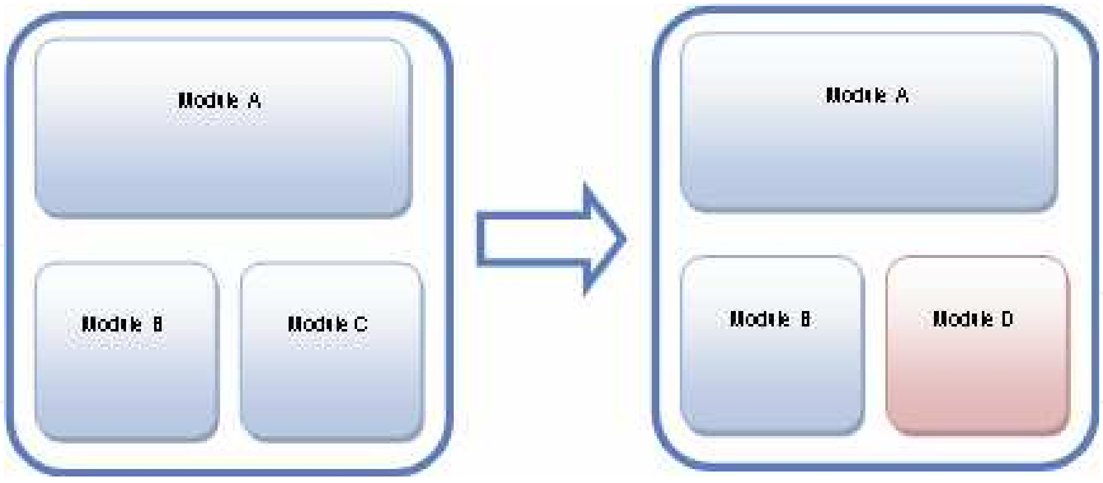


Fig. 2. Switching from Module C to Module D.

The two first reconfiguration types are important because they map the chip modification to save area (module removing) and to change functionality (module switching). The module partitioning is important to enable the same functionality be partitioned into different modules scheduled on time. In this way, we have the three basic benefits from partial and dynamic reconfiguration, save area, change functionality and time partitioning, other benefits are consequences of these.

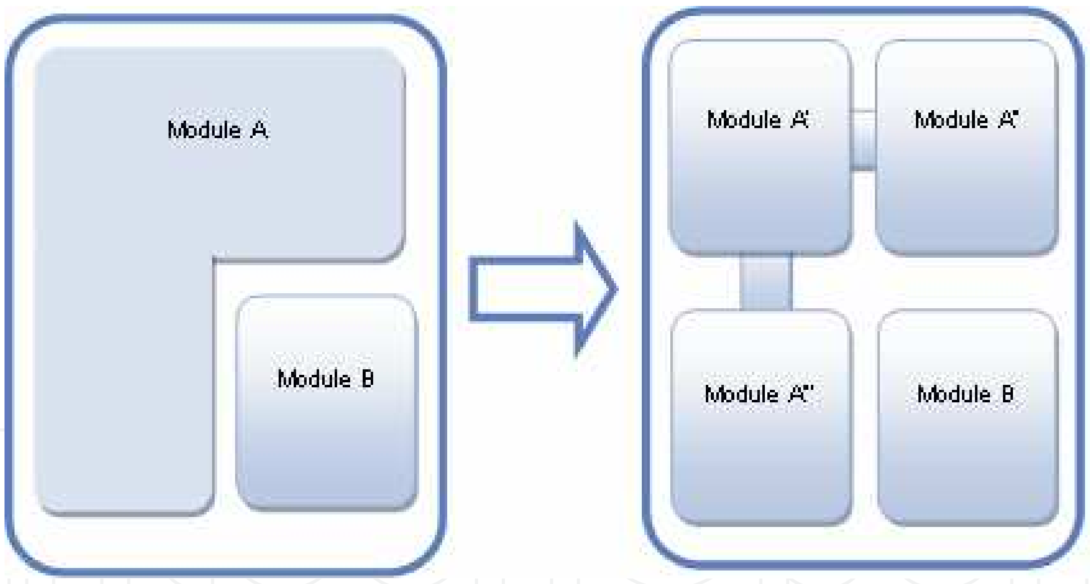


Fig. 3. Module partitioning into three different modules of same functionality.

3. The methodology

The methodology created to simulate dynamic reconfiguration is based on changing the execution mechanism of discrete-event simulators. The simulator must check every module before executing it, verifying if they were deactivated before. In the affirmative case the module must not be executed.

Fig. 4 presents a general simulator module based on events and organized in modules and processes, used mainly for digital hardware systems simulation. Each module can implement one or more processes, which execute the task. The processes have a sensitivity

list each, indicating which events they are sensitive to. A process is executed on a simulation cycle if one event registered on its sensitivity list occurs during that specific cycle.
In the example of Fig. 4, the event E3 could represent the clock signal modification, and as we can see, every process is sensitive to it; each clock signal will trigger every processes to be executed.
The scheduler is part of the simulator kernel, and decides the execution sequence for each cycle. If event E1 is scheduled, for example, it will be searched on the processes sensitivity lists, and be found on processes 1 and 3, which belong to modules A and B, respectively.
The simulated time is formed by a sequence of simulation cycles. At each cycle, one or more events can occur. In case of no event occurs during a cycle, the simulation clock advances and none activity is performed, making the simulation faster. The simulation performance depends directly on sensitivity lists. The more events the lists have, more probable is a process to be executed and new cycles to be created, which costs hardware processing.
Back to dynamic reconfiguration, a not configured module can be defined as a never-executed module, not depending on occurred events, neither on its sensitivity list. On the same way, not configured modules can be reconfigured during simulation just by allowing its normal execution based on events.
Our methodology lies on the interception of the execution signals generated from the simulator to the modules, making that not configured modules never receive those signals. Conceptually, we adopted the module blocking instead of process blocking, as a module normally represents a hardware functionality unit.

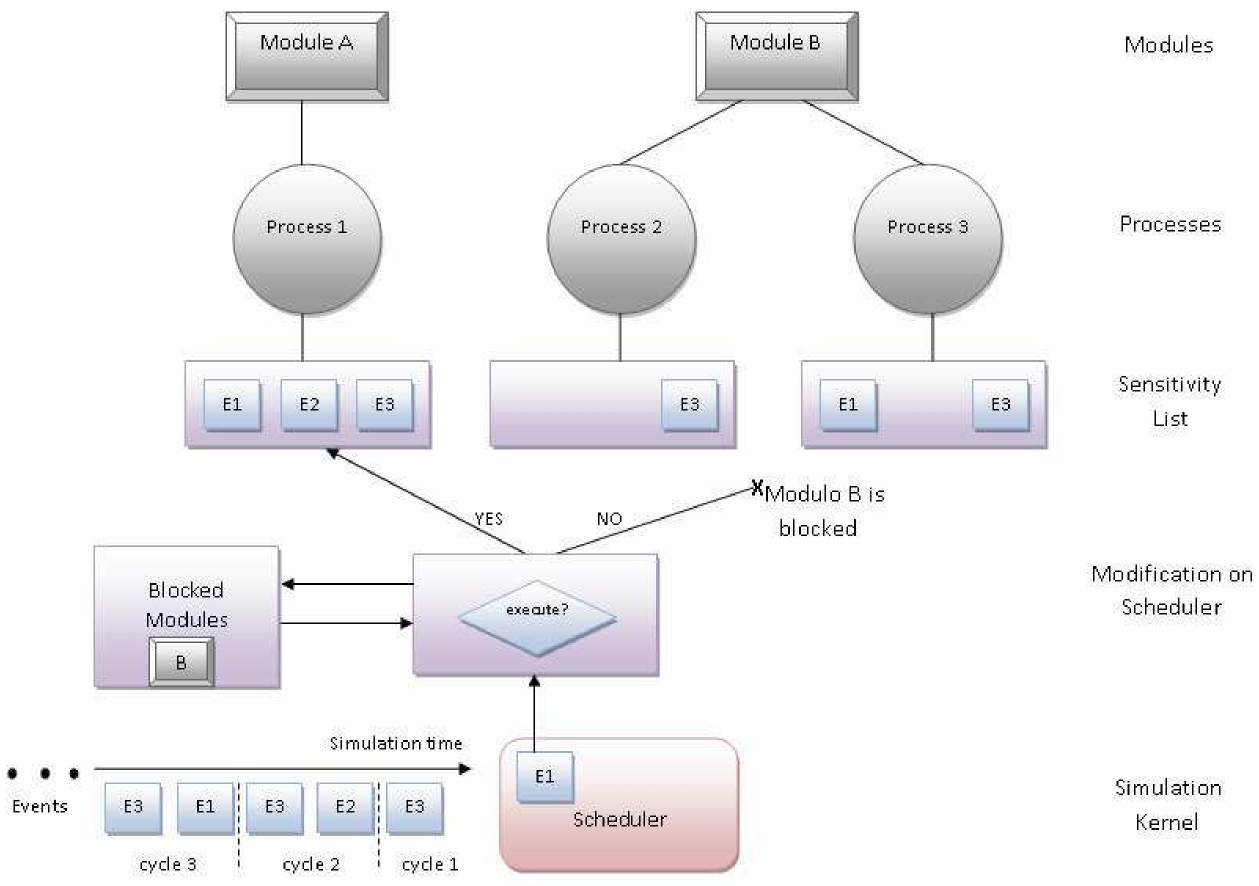


Fig. 4. Modified simulator to block modules not more configured on system.

Fig. 4 presents the modification that should be done aiming at interception of execution signals to not configured modules. Our strategy was implemented by creating a blocked modules list. Instead of immediately executing the processes sensitive to an event we propose the blocked modules list to be checked before its execution. The process will be executed only if the module which it belongs to does not appears in the list. For example, on Fig. 4, the Module B was found on blocked modules list. For that reason, the Process 3 was not executed, although it is sensitive to event E1.
Using our methodology the implementation of dynamic reconfiguration on a simulator can be performed just by managing the blocked modules list, adding some kind of reference to the modules that should not be configured and removing it to reconfigure the module.

4. Adding dynamic reconfiguration simulation to SystemC

In order to implement the methodology using a functional simulator, SystemC was selected. We adopted a bottom-up approach adding functions to activate and deactivate modules by the programmer during simulation. SystemC is open-source, free and enables the modeling and simulation at TLM and RTL using Object-Orientation concepts (Grotker et al., 2002). It does not allow deactivation of modules during simulation, but as an open-source tool, it is a great candidate for our methodology application. The Adriatic project (Qu et al., 2004) also uses SystemC at transaction level (TLM), but it does not simulate the dynamic behaviour of the modules during simulation. On the other hand the OSSS+R project (Schallenberg et al., 2006) simulates the dynamic reconfiguration of SystemC modules using heritage and polymorphism. It implements a SystemC language extension which allows the switching of modules inherited from the same super class. This top-down approach does not allow the simulation of RTL systems, neither its application to other not Object-Oriented simulators.
The strategy is implementing two special functions for activating and deactivating modules during simulation named *dr_sc_turn_on* and *dr_sc_turn_off*, respectively. Both were written modifying the SystemC kernel source code. Figure 6 presents the added functions declarations in the *sc_simcontext.h* SystemC header file. The two routines *dr_add_constraint* are used to store modules attributes, like the chip area occupation by the module and the reconfiguration delay, always present when a module is configured on chip. The *extern* key-word indicates that the routine can be called outside the *sc_context* class. In other words, those functions can be called by user code on regular simulations.

```
extern void dr_sc_turn_off(std::string module_name);
extern void dr_add_constraint(std::string module_name, int area);
extern void dr_add_constraint(std::string module name, int area,
                              sc time reconfDelay);
extern void dr_sc_turn_on(std::string module_name);
```

Fig. 5. Main routines added to SystemC library (sc_simcontext.h)

In Fig. 6 is presented how the functions were implemented in sc_simcontext.cpp SystemC kernel file. A linked list is used to store the names of the modules that must be not executed (not configured). The routine *dr_sc_turn_off* add the module name to the list, while the *dr_sc_turn_on* remove the module from the list, allowing it to be executed (reconfigured). Another list is keep to store the modules constraints (chip area and reconfiguration delay). This list is required when the *dr_add_constraint* function is called. In this case, constraints are added to the list and cannot be removed, just overwritten. The chip area of each module is used for chip occupation analysis normally performed after simulation. Such analysis is important to figure out how effective the application of dynamic reconfiguration on chip was.

```
void dr_sc_turn_off(std::string module_name){
    sc_get_curr_simcontext()->dr_add_config(module_name);
}

void dr_sc_turn_on(std::string module_name){
    sc_get_curr_simcontext()->dr_remove_config(module_name);
}

void dr_add_constraint(std::string module_name, int area){
    sc_get_curr_simcontext()->dr_addConstraint(module_name,area);
}

void dr_add_constraint(std::string module_name, int area, sc_time delay){
    sc_get_curr_simcontext()->dr_addConstraint(module_name,area,delay);
}
```

Fig. 6. The added routines from Figure 6 implemented in sc_simcontext.cpp file.

The details of the routines to manipulate the linked lists are presented on Fig. 7. Adding a module name into the configuration list (*dr_add_config* function) is not a problem. The module name is simply added into the list. But, the *dr_remove_config* just remove the module name from the list if the reconfiguration delay for that module has expired, and the first call of this function is considered just a removing request. Therefore, before removing the module name, the delay is compared with the elapsed time since the removing request.

```
void sc_simcontext::dr_add_config(std::string module_name){
    configList->addConfig(module_name,m_curr_time,0,
                          constraintList->getReconfDelay(module_name));
}

void sc_simcontext::dr_remove_config(std::string module_name){
    sc_time delay = configList->getReconfDelay(module_name);
    if(delay > sc_time(0,SC_NS)){
       configList->setActionTime(module_name,delay + m_curr_time);
       configList->request_remove(module_name, true);
    }
    else
       configList->removeConfig(module_name);
    }

void sc_simcontext::dr_addConstraint(std::string module_name, int area){
    constraintList->addConfig(module_name,m_curr_time,area);
}

void sc_simcontext::dr_addConstraint(std::string module_name, int area,
                                      sc_time reconfDelay){
    constraintList->addConfig(module_name,m_curr_time,area,reconfDelay);
}
```

Fig. 7. Implementation of the new routines.

Now the SystemC execution properly can be performed. This execution is made at *sc_simcontext* class by the *crunch* method. The modified code can be seen on Fig. 8. Initially in line 3 the method *pop_runnable_method* returns the *sc_method_handle* to the next method to be executed at simulation. The modifications aim at the execution avoidance of methods from *ConfigList* and store the execution history of each module in a logging file. The *fout* object is responsible to print every event on log file. The blocked modules are represented in log file with and "X" (lines 18 and 20), and when the module is executed, the module area is printed on file instead (lines 15 and 28).
The three conditionals on lines 10, 11 and 12, check whether the module should be executed or not. Initially is checked whether module name is on *ConfigList* (line 10), and then whether the *request_remove* was called for the module (line 11), finishing the verification checking whether the reconfiguration delay was already elapsed (line 12). If all verifications are true, the module is removed from *ConfigList* (line 13) and finally executed (line 14). Following the process, the module area is printed on log file (line 15). Case any conditional returns false, a "X" is printed on log file representing execution blocking.

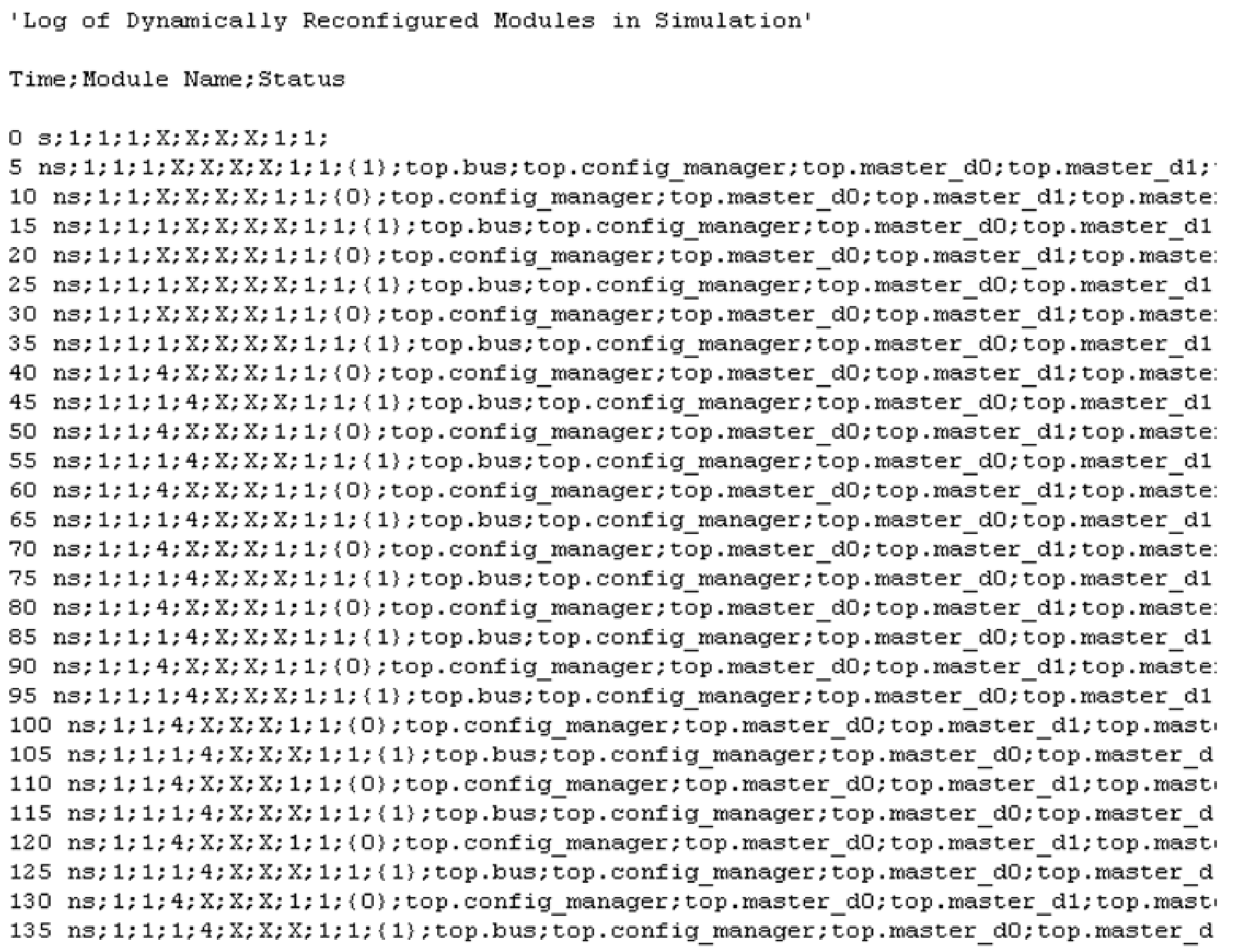
```
1. while( true ) {
2.  // execute method processes
3.  sc_method_handle method_h = pop_runnable_method();
4.  while( method_h != 0 ) {
5.    try {
6.      if(m_curr_time > sc_time(0,SC_NS)){
7.         str += get_method_name(method_h);
8.         str += ";";
9.      }

10      if(configList->exists(get_method_name(method_h))){
11         if(configList->isOff(get_method_name(method_h)))
12            if(configList->getActionTime(get_method_name(method_h)) <= m_curr_time){
13              configList->removeConfig(get_method_name(method_h));
14              method_h->execute();
15              fout << constraintList->getArea(get_method_name(method_h)) << ";";
16            }
17            else
18              fout << "X;";
19         else
20           fout << "X;";
21         }
22      else{
23         method_h->execute();
24         fout << constraintList->getArea(get_method_name(method_h)) << ";";
25      }
26   }
27   catch( const sc_report& ex ) {
28      ::std::cout << "\n" << ex.what() << ::std::endl;
29      m_error = true;
30      return;
31   }
32   method_h = pop_runnable_method();
33  }
```

Fig. 8. SystemC crunch routine, responsible for executing every module in simulation.

4. Execution logging

As detailed before, every simulation cycle is logged on a file. A fragment of the log file is presented on Figure 10. Each line on the log file stores the simulation cycle timestamp, the modules occupation area and the respective module names. If a module is not configured at that time, and "X" is stored instead of its chip area. All information is stored on CSV format (*Comma-separated values*).



```
'Log of Dynamically Reconfigured Modules in Simulation'

Time;Module Name;Status

0 s;1;1;1;X;X;X;X;1;1;
5 ns;1;1;1;X;X;X;X;1;1;{1};top.bus;top.config_manager;top.master_d0;top.master_d1;
10 ns;1;1;X;X;X;X;1;1;{0};top.config_manager;top.master_d0;top.master_d1;top.maste
15 ns;1;1;1;X;X;X;X;1;1;{1};top.bus;top.config_manager;top.master_d0;top.master_d1
20 ns;1;1;X;X;X;X;1;1;{0};top.config_manager;top.master_d0;top.master_d1;top.maste
25 ns;1;1;1;X;X;X;X;1;1;{1};top.bus;top.config_manager;top.master_d0;top.master_d1
30 ns;1;1;X;X;X;X;1;1;{0};top.config_manager;top.master_d0;top.master_d1;top.maste
35 ns;1;1;1;X;X;X;X;1;1;{1};top.bus;top.config_manager;top.master_d0;top.master_d1
40 ns;1;1;4;X;X;X;1;1;{0};top.config_manager;top.master_d0;top.master_d1;top.maste
45 ns;1;1;1;4;X;X;X;1;1;{1};top.bus;top.config_manager;top.master_d0;top.master_d1
50 ns;1;1;4;X;X;X;1;1;{0};top.config_manager;top.master_d0;top.master_d1;top.maste
55 ns;1;1;1;4;X;X;X;1;1;{1};top.bus;top.config_manager;top.master_d0;top.master_d1
60 ns;1;1;4;X;X;X;1;1;{0};top.config_manager;top.master_d0;top.master_d1;top.maste
65 ns;1;1;1;4;X;X;X;1;1;{1};top.bus;top.config_manager;top.master_d0;top.master_d1
70 ns;1;1;4;X;X;X;1;1;{0};top.config_manager;top.master_d0;top.master_d1;top.maste
75 ns;1;1;1;4;X;X;X;1;1;{1};top.bus;top.config_manager;top.master_d0;top.master_d1
80 ns;1;1;4;X;X;X;1;1;{0};top.config_manager;top.master_d0;top.master_d1;top.maste
85 ns;1;1;1;4;X;X;X;1;1;{1};top.bus;top.config_manager;top.master_d0;top.master_d1
90 ns;1;1;4;X;X;X;1;1;{0};top.config_manager;top.master_d0;top.master_d1;top.maste
95 ns;1;1;1;4;X;X;X;1;1;{1};top.bus;top.config_manager;top.master_d0;top.master_d1
100 ns;1;1;4;X;X;X;1;1;{0};top.config_manager;top.master_d0;top.master_d1;top.mast
105 ns;1;1;1;4;X;X;X;1;1;{1};top.bus;top.config_manager;top.master_d0;top.master_d
110 ns;1;1;4;X;X;X;1;1;{0};top.config_manager;top.master_d0;top.master_d1;top.mast
115 ns;1;1;1;4;X;X;X;1;1;{1};top.bus;top.config_manager;top.master_d0;top.master_d
120 ns;1;1;4;X;X;X;1;1;{0};top.config_manager;top.master_d0;top.master_d1;top.mast
125 ns;1;1;1;4;X;X;X;1;1;{1};top.bus;top.config_manager;top.master_d0;top.master_d
130 ns;1;1;4;X;X;X;1;1;{0};top.config_manager;top.master_d0;top.master_d1;top.mast
135 ns;1;1;1;4;X;X;X;1;1;{1};top.bus;top.config_manager;top.master_d0;top.master_d
```

Fig. 9. A fragment from an execution log file.

The log file can easily be exported to calculations softwares and the system behavior can be seen in table format, furthermore, graphics can be made. Fig. 10 presents an example of a graphic representing the chip area utilization over the time. On this example, some modules are not configured during some time intervals, making the total chip area varying from 7 to 12 area units (hypothetical unit).

Using this strategy, conventional systems can also have their execution log analyzed and candidate modules for partial reconfiguration can be detected. Therefore, the log analysis can be used as the first step for system behavior study.

5. Experiments and results

In order to apply our methodology to model and simulate dynamically reconfigurable hardware systems, two case studies were developed.

The first work was the modelling and simulation of a research project for Daimler-Crysler in collaboration with the University of Karlsruhe in Germany (Becker & Vorbach, 2003). The objective is to simulate a dynamically reconfigurable hardware, which controls some eight

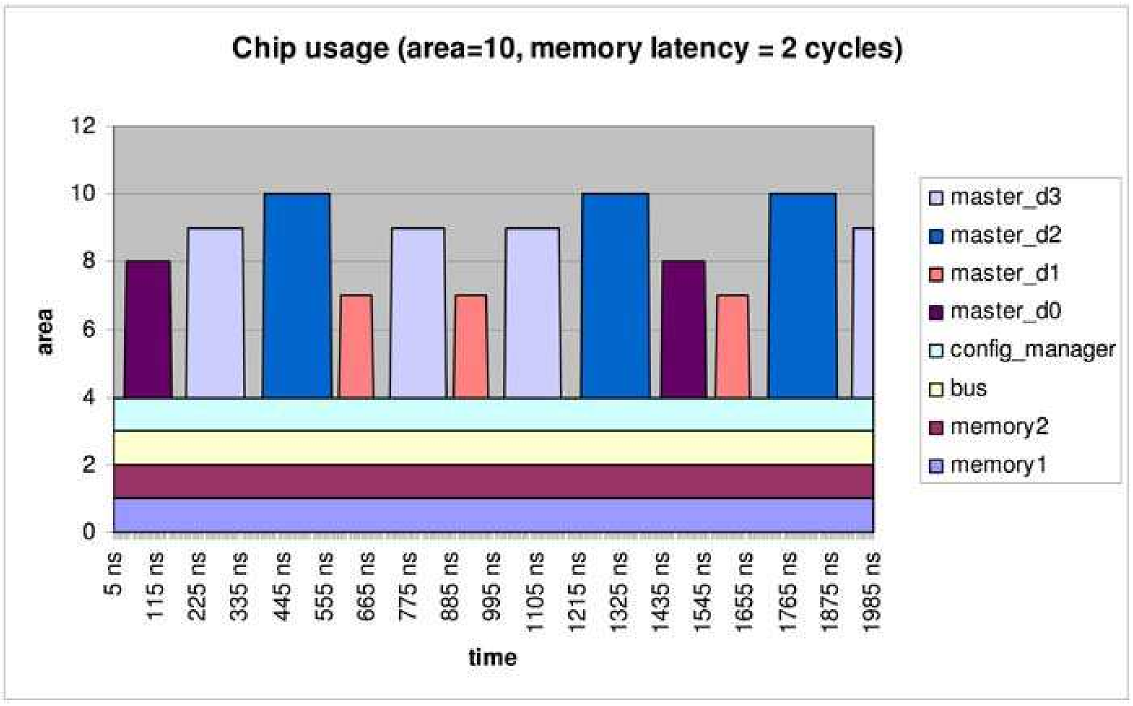


Fig. 10. Chip area usage generated from an execution log file.

inner cabin devices on-demand, four windows, two seats, one internal mirror and one controller for the lights. If the user requests a certain service, the corresponding hardware unit is configured and initialized in an unoccupied slot within the dynamic reconfigurable area of the FPGA system (see Fig. 11). The results of this work were published in *IEEE Computer Society Annual Symposium on Emerging VLSI Technologies and Architectures (ISVLSI'2007)* in Porto Alegre (Brito et al., 2007).

In this example, a maximum number of four applications can be executed in parallel. This hardware constraint enables the reduction of the number of different electronic hardware control units within a car, hence saves space, power consumption and costs. The justification of hardware implementation can be easily demonstrated, when considering heavy CAN traffic, where traditional microprocessor based systems reach their limits.

The example application is implemented on a Xilinx Virtex-II FPGA (XC2V3000). To get an overview of the complete system, Fig. 11 shows a block schematic containing all main integrated components. A MicroBlaze Soft-IP controller from Xilinx is used. A detailed description of the run-time system and the tasks of the MicroBlaze controller can be found in (Huebner et al., 2004).

The FPGA of the Virtex-II series also provides an internal configuration access port (ICAP) that allows reconfiguration without the need of external wiring. The partial bitstreams for the modules are stored in an external flash memory. The run-time system accesses them by sending start address and end address to a decompressor module on demand. A further start command enables the decompressor, which reads the compressed bitstream from the flash memory in order to write the configuration code through the ICAP interface to the internal configuration memory of the FPGA.

While it is being processed, the controller and all other modules are able to continue the execution of their tasks. A signal from the decompressor reports the end of the configuration process, which indicates that the service is ready for use. The complete system is connected via a CAN interface to its environment.

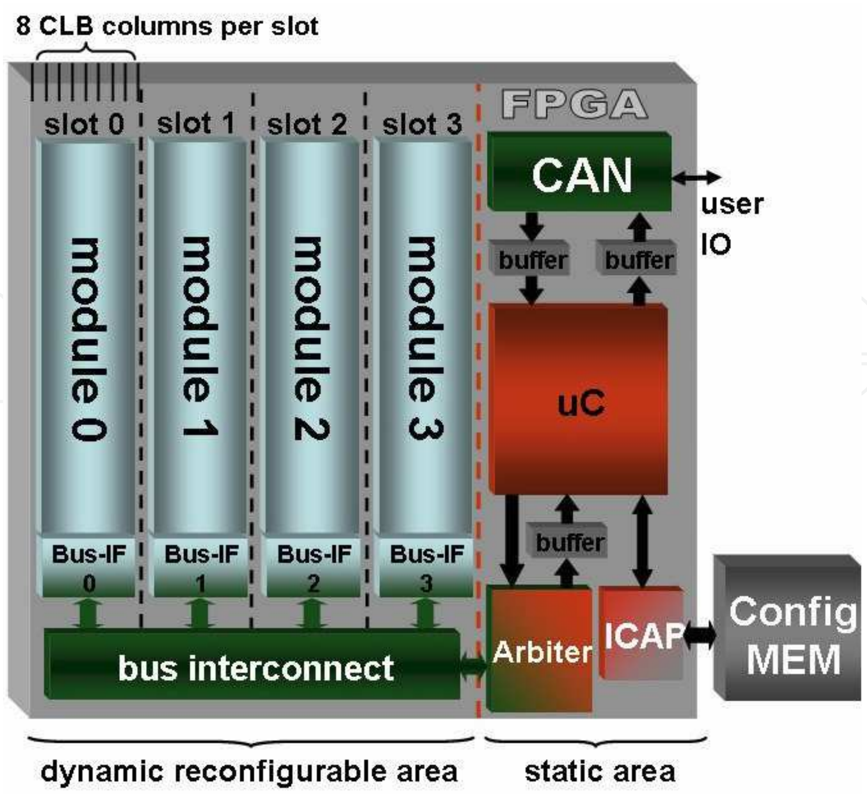


Fig. 11. Architecture of the automotive system.

The experiments show that if the FPGA's dynamic area is constrained to 8 CLB columns which are equal to 1 application slot, the average response time is about 1000 times larger than the client timing constraint, which is 100ms. On the contrary, a system owning 64 CLB columns, where all eight applications can be configured at the same time, the average response time satisfies the timing constraint. However, the area usage is far from being reasonable or efficient.

Actually the real hardware implementation (as represented by Fig. 11) uses 32 CLB columns. In this case, the simulations show that the system's average response time is shorter than 100ms if the request rate is set to maximum 1 per second. A larger rate implies a system stall for a specific period of time. The problem arises mostly after the fifth request, when no slot is temporarily available.

The response time with 32 configurable columns satisfies most use cases, although with 24 columns (which mean 3 applications per time) it may be sufficient for non-critical applications. It could not respond instantly, for example, if a window were closed with somebody having his hand in between.

The second example implements a general purpose simulator for processors, called PReProS (A General Purpose Partially Reconfigurable Processor Simulator), whereas this technique supports run-time reconfiguration (Brito et al., 2007). Such technique uses high-level representations to model and simulate the reconfiguration, giving the opportunity to designers to foresee the dynamic behavior of your system before the hardware is going to be implemented for the target architecture, or even before the system specification in HDL, if desired.

Considering the simulation of dynamic and partially reconfigurable systems, a couple of steps should be done, like the target architecture specification, the definition of necessary hardware resources and the designing of the applications. The presented approach aims at writing a reusable parametrizable SystemC program able to model and simulate real target processor architectures. For example, coarse-grained like XPP (Becker et al., 2003) which consists of configurable ALUs communicating via a packet oriented, automatically synchronized communication network. Further, fine-grained architectures, like standalone FPGAs and embedded FPGAs, which have the well known FPGA behavior, or any other processor, running any kind of application.

The goal is to parameterize the individual processor's characteristics in such a general way that all kind of processing element can be fully described using this set of parameters. The main features that have to be considered here are the clock frequency, properties of the data and configuration ports, and the number of chip area available on chip. In the same way, the applications' properties can be set by the frequency, needed ports, data width and number of configured area units. When using this simulator the designer just have to set the parameters and implement its own blocks to configure the applications and exchange data with the PReProS.

On Fig. 12 it is possible to see the amount of used resources of the XPP simulated chip when five different applications were scheduled. XPP was simulated containing 144 ALUs. In this way more parallel configurations could be simulated. The free area is marked by the darker area in the figure. By investigating these results, the best parallel performance and hence the best processing power and efficiency of the simulated processor area can be achieved. It helps the designer to reevaluate his/her algorithms and implementation strategy, or if the selected architecture should be changed to better target his needs.

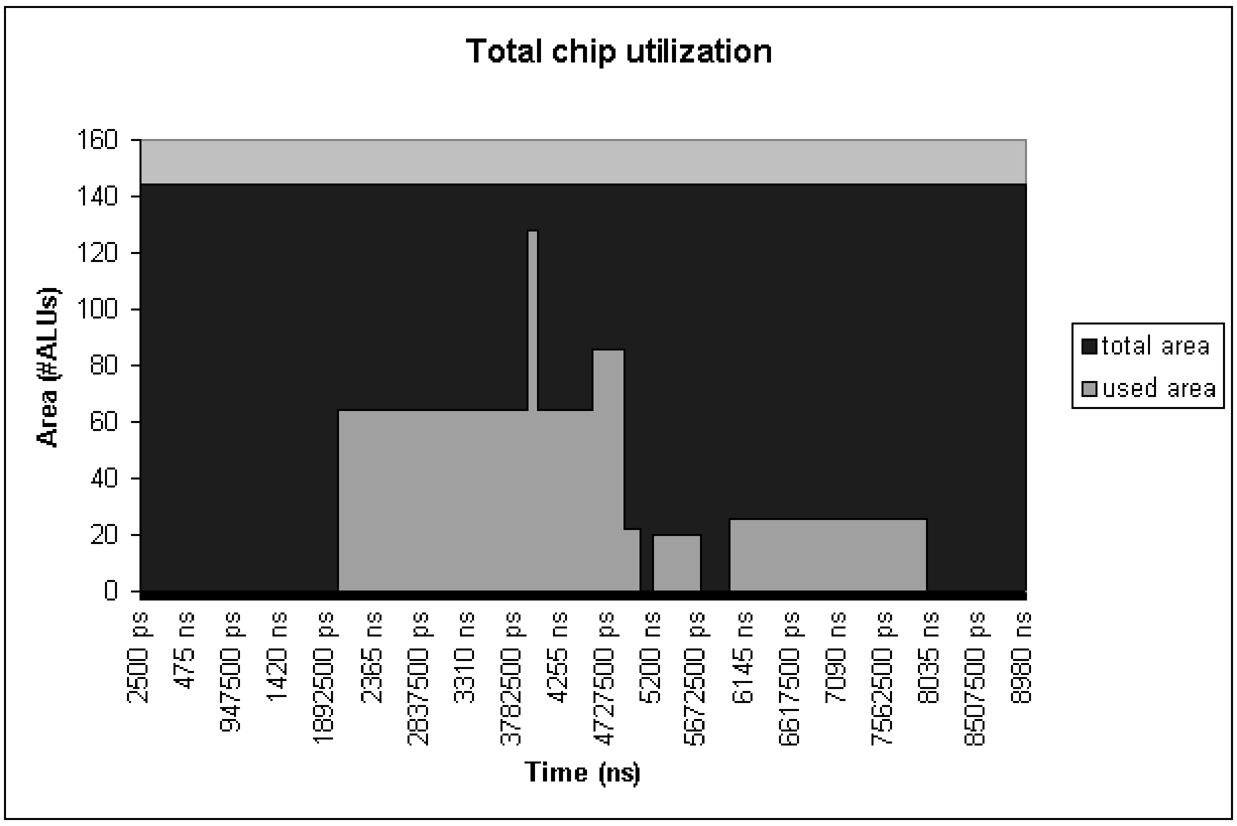


Fig. 12. Total chip area utilization by PRePros

6. New design-flow with partial and dynamic reconfiguration

An important aspect is the integration of this modeling and simulation technique into the design flow. It is desired to achieve this in a plug and play manner. To provide such lightweight integration, the SystemC (www.systemc.org) description language is used. The capabilities are presented as an easy to use API and can be applied to any system, which is described in SystemC. Fig. 13 presents a typical SystemC based design flow. Usually, the same approach is used twice, to develop both, statically and dynamically reconfigurable systems. The absence of specific techniques and tools would turn such development into an arduous and costly task.
During hardware verification, it is quite common to iterate several times within the design cycle, thus returning to the TLM and RTL model. Our technique aims at reducing these verification cycles and, as a result, decreasing development time.
There are other efforts to provide similar functionalities using SystemC. However, they are mostly TLM-based (like OSSS+R project (Schallenberg et al., 2004)) including operation restrictions, or do not focus on simulation (like Adriatic project (Qu et al., 2004)). The presented approach attacks the same problem in a more general way. Any module can be removed, added or switched at simulation-time.
Aiming at decreasing design time, an extension of the common SystemC based design flow is proposed. The modeling and simulation of dynamic and partial reconfiguration is aggregated, resulting in a modified design flow, as shown in Fig. 13.
The dynamic behavior at TLM or RTL is performed by specific instructions. The designer decides about a proper location in his code. An interesting way is an implementation in one or more separated models, which centralizes the dynamic behavior of the system. These modules can then be realized as dedicated blocks that control and schedule run-time configuration.

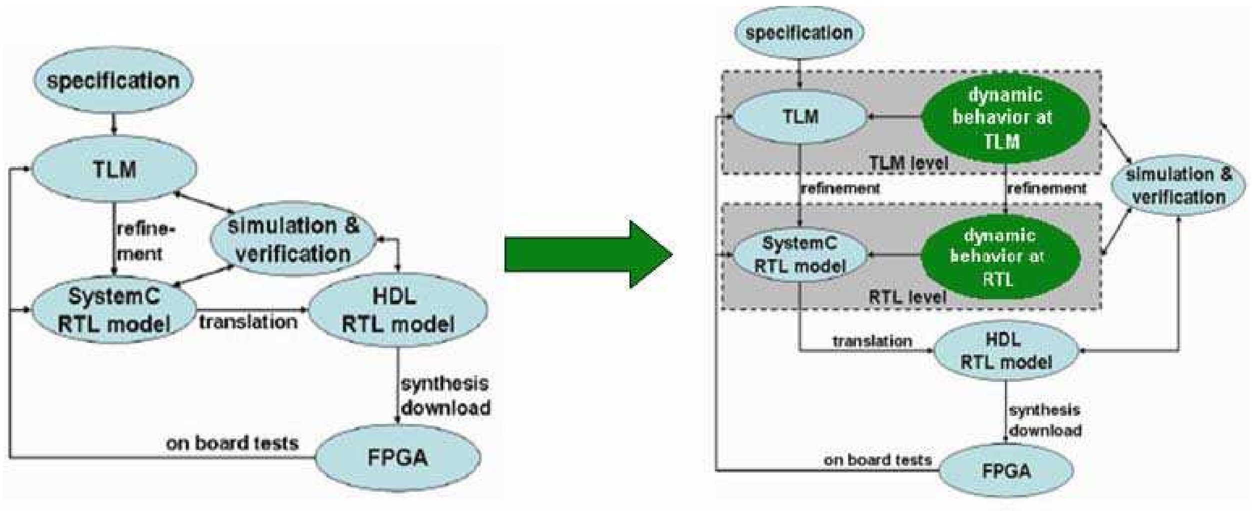


Fig. 13. Typical design-flow using SystemC adapted for dynamic reconfiguration.

7. Proof of concept

In order to validate the methodology, we should proof that all three types of dynamic reconfiguration operations defined on Section 2 can be modeled and simulated by our SystemC modified version. In general, the strategy used to model partial reconfigurable system with SystemC is based on, first declares and connects all modules that will be part of the system at some time. Fig. 14 shows how the module substitution should be done. It replaces the *sc_module_C* by the *sc_module_D*. Initially both modules are present in the system, but just the module C is configured. At the second moment, the module C is deactivated by calling *dr_sc_turn_off ("moduleC")* and the module D is configured by calling *dr_sc_turn_on ("moduleD").*

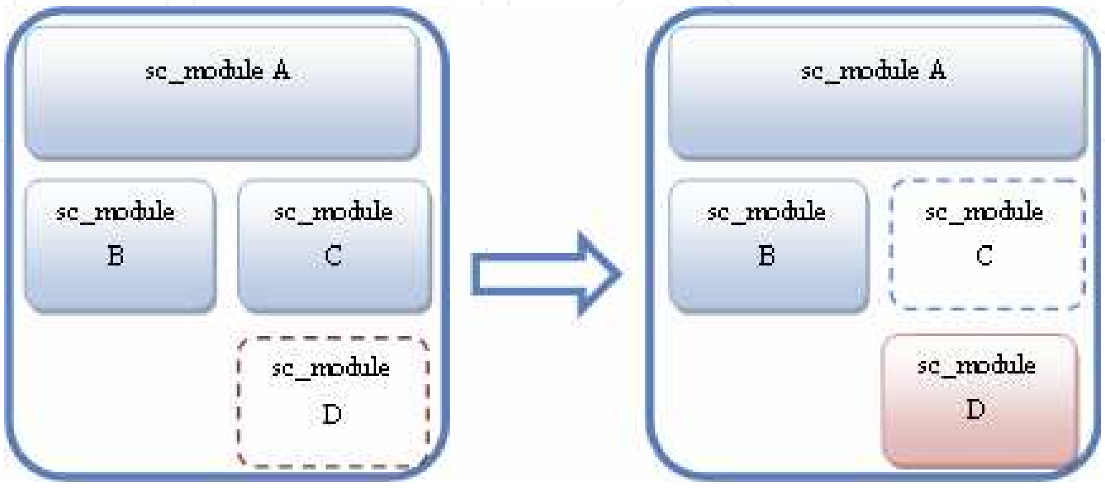


Fig. 14. Implementation of the first dynamic reconfiguration type (switching).

On the second type of reconfiguration operation, a module should be removed from the system. As can be seen on Fig. 15, at the first step every modules were instantiated on simulation, and at the second the module C was deactivated by calling *dr_sc_turn_off ("moduleC").*

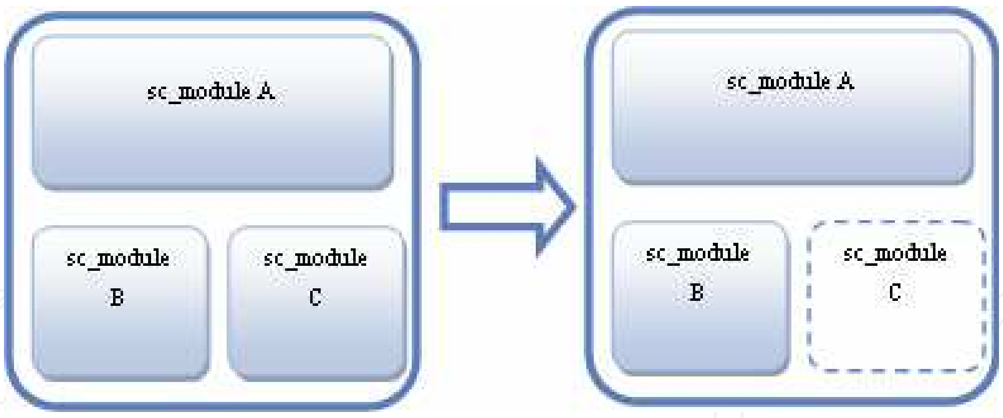


Fig. 15. Implementation of the second dynamic reconfiguration type (removing).

For the third partial reconfiguration operation (see Fig. 16) is necessary instantiate the complete module (*sc_module A*) at the same time with all the sub modules (modules A′, A′′ and A′′′) that execute the module A functionality partitioned in time. The sub-modules are deactivated at the first time and at the second moment the module A is deactivated and the sub modules are configured.
These three demonstrations show that using the methodology is possible to simulate the three basic dynamic reconfiguration operations, the module switching, removing and partitioning. We believe that each simulator able to simulate these three operations is able to simulate any dynamically (and partially) reconfigurable system.

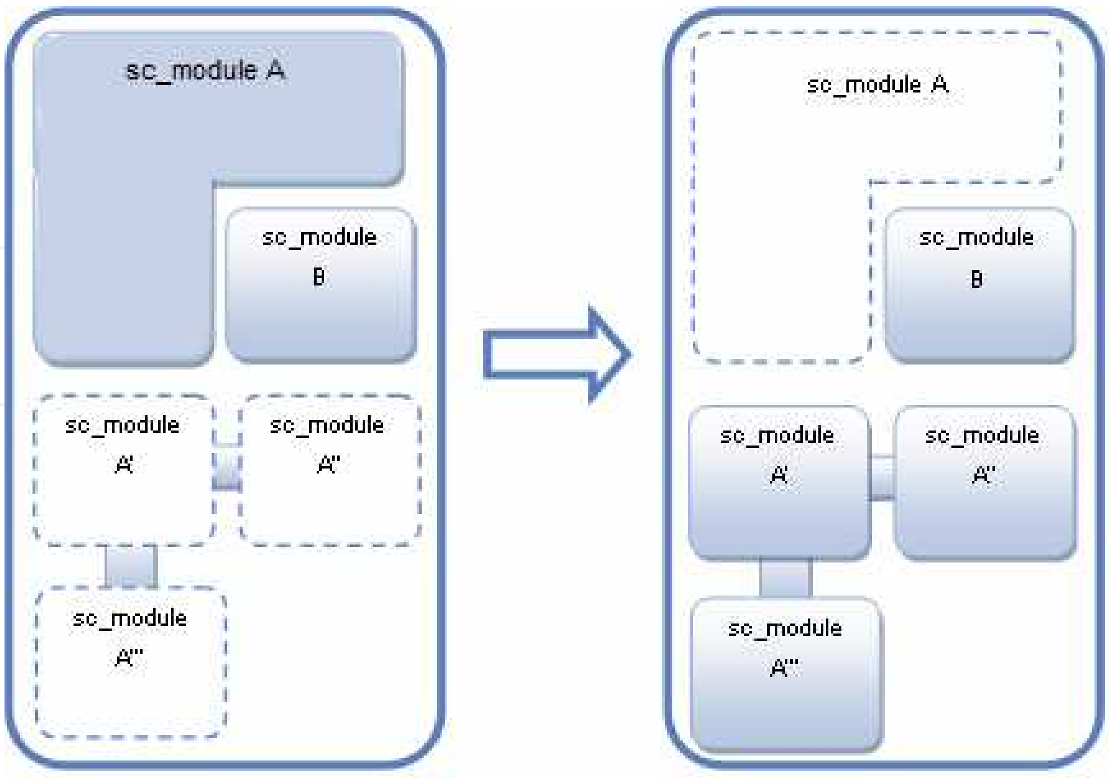


Fig. 16. Implementing the third type of reconfiguration (partitioning).

8. Simulator performance

The feasibility of the methodology was demonstrated for the first time in (Brito et al., 2006). In (Brito et al., 2007) an automotive application was simulated and the dynamic reconfiguration benefits could be visualized using the logging feature. In (Brito et al., 2007) a general purpose simulator was created using the modified SystemC, in order to simulate processors with dynamic reconfiguration features like some FPGAs and coarse-grained chips.

These works were designed at TLM, so the performance of using the modified SystemC was not significantly low. Our experiments demonstrated some performance limitations with RTL simulations. The simulator performance was tested simulating an MPEG-4 decoder (Rocha et al., 2006). Initially the system was modeled used SystemC RTL and brought to chip synthesis and silicon fabrication. The decoder implements the *Simple Profile Level 0* from MPEG-4. The decoder architecture contains the project of a personalized hardware to the bitstream decoding, *Variable Length Code* (VLC), texture decoding, movement compensation and color spaces conversion. The experiments on hardware demonstrate that 30 frames per second were decoded (Rocha et al., 2006).

A 16 frames video was simulated in two different runs. For the first run the original SystemC version 2.1.1 without modification was used. For the second run the modified SystemC of the same version was used. In both cases, no dynamic reconfiguration was used. The results show that the modified simulator presented a three times slower simulation than the SystemC without modification (see Table 1).

Using the modified SystemC, the Config List is checked at each simulation cycle. This checking causes a negative impact to the simulation time, as the list is completely analyzed at each cycle. We believe that the simulation time increase is tolerable considering the advantage to be able to simulate dynamic reconfiguration both at RTL and TL abstraction level.

Simulator	Simulation time
Traditional SystemC	12m56.630s
Modified SystemC	36m34.334s

Table 1. Performance of the modified SystemC in RTL (MPEG-4 decoder example).

9. Methodology extension

In general, the methodology principles applied in partial reconfiguration consists in turning off a sub-module "A" before of configuration of the sub-module "B" in the area previously occupied by "A" and then turning on the sub-module "B". Analyzing the turning on and off principles of the sub-modules, these principles are similar to those adopted by the technique of power gate, differentiating which in technique power gate turning on and off the same module. This section will be shows the work based on reusability of the methodology to modify the SystemC simulator, the purpose is to simulate power gate design in RTL (Silveira et al., 2009).

9.1 Overview power-gate

Power gate strategy is based on adding mechanisms to turn off blocks within the SoC that are not being used, the act of turning off and on the block is accomplished in appropriate time to achieve power saving while minimizing performance impact (Keating et al., 2007).
When the event of turning off happens, the energy savings is not instantaneous due to thermal issues of the previous activity and the nature of technology is not ideal for power gate. In the event of turning on the block requires some time that cannot be ignored by the system designer for the block to retake the activity (Keating et al., 2007). Fig. 17 shows an example of the activity of a block with power gate implemented.

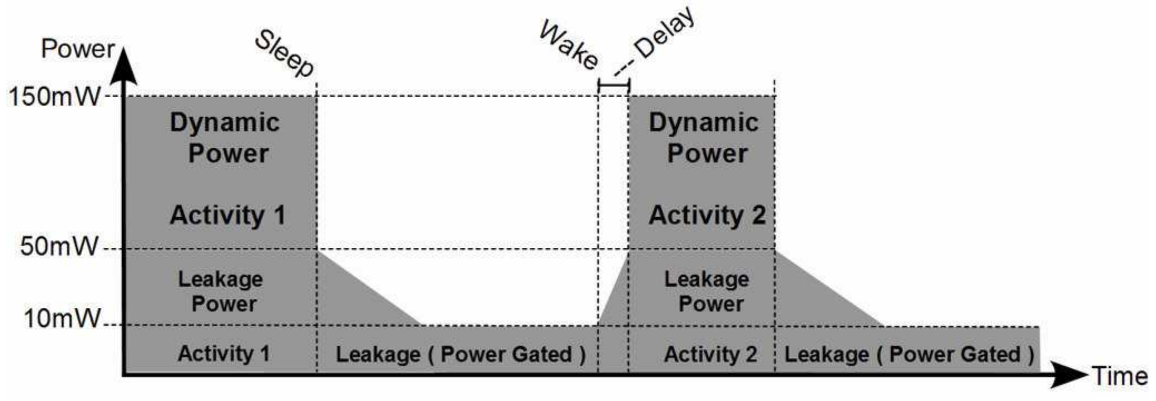


Fig. 17. Profile with Power Gating (Keating et al., 2007)

Differently of a block that is always active, a power-gate block is powered by a power-switching network that will supply VDD or VSS power gate block, the CMOS (Complementary Metal–Oxide–Semiconductor) switches are distributed within or around the block. Control of CMOS switches is accomplished by a power gating controller. In some cases it is necessary to retain the state of the block during the turned off period to restore the state when it is turned on. This restraint is implemented using special flip-flops. Figure 18 shows the diagram with the structure of the SoC with power gate.

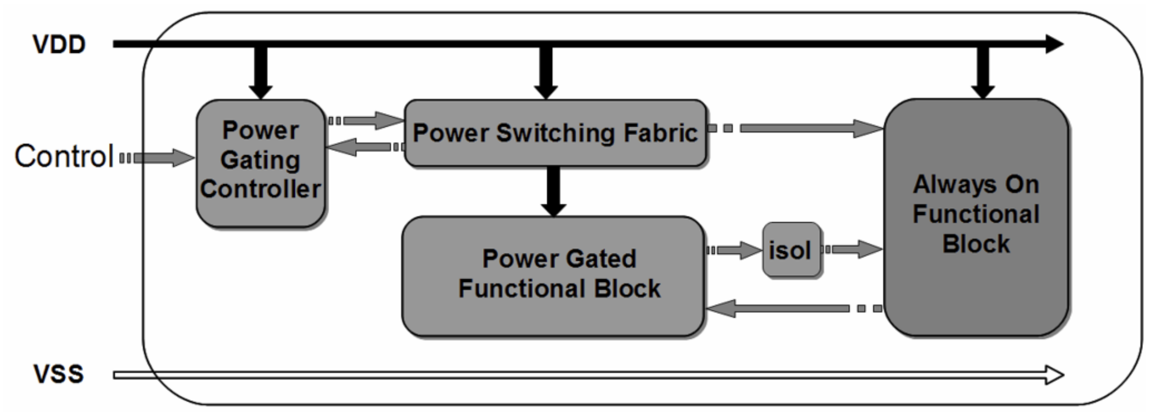


Fig. 18. Block Diagram of a SoC with Power Gating (Keating et al., 2007)

9.2 Simulator implementation

In order to implement the functional verification of low power design using a functional simulator, a similar approach developed to simulate partial and dynamic reconfiguration (Brito et al., 2006; Brito et al., 2007) was selected. This is a bottom-up approach adding functions to activate and deactivate modules by the programmer.
The strategy implements two new special functions for turning on and off modules during simulation named *sc_lp_turn_on* and *sc_lp_turn_off*, respectively. These functions were written modifying the SystemC kernel source-code. The routine *sc_lp_add_constraint* was also created and is used to store modules attributes about their energy consumption and the turn-on delay, always present when a module is re-activated on chip. Table 2 presents how the functions signatures in sc_simcontext.h SystemC kernel file.

extern void sc_lp_turn_on(std::string module_name);
extern void sc_lp_turn_off(std::string module_name);
extern void sc_lp_add_constraint(std::string module_name, sc_time wakedelay);

Table 2. Functions declarations

A linked list is used to store the names of the modules that must be not executed (turn-off). The routine sc_lp_turn_off adds the module name to the list, while sc_lp_turn_on removes the module from the list, allowing it to be executed (activity). Another list is kept to store the module constraints (wake delay and energy consumption). This list is required when the sc_lp_add_constraint function is called. In this case, constraints are added to the list and cannot be removed, just overwritten. The extern key-word indicates that the routine can be called outside the sc_context class. In other words, those functions can be called by user code on regular simulations.

9.3 Functional verification

VeriSC methodology adopts projects with hierarchy concept, therefore a project can be divided into parts to be implemented and verified (Silva & Melcher, 2005). BVE-Cover library was chosen to accomplish the functional verification with coverage of the design. Several simulations were performed with different versions of SystemC simulator and design:

- **SC + DV1:** At this stage were used the original SystemC version 2.2.0 and the first implementation of the design.
- **SC–LP + DV1:** At this stage were used the new SystemC-LP functions added and the first implementation of the design.
- **SC–LP + DV2:** At this stage were used the new SystemC-LP version and the second implementation containing the power gate design.

9.4 Results

Several results were extracted (Silveira et al., 2009), but with respect to reusability of the methodology we can highlight, (1) was possible to simulate low power design in RTL, and during the simulation we can verify the power gate principles operating; (2) the simulator performance loss, which a negative point, fact occurred due to the adoption of the strategy used in the dynamic reconfiguration simulator. Fig. 19 shows a graphic with the different simulators performance. The first simulation time was measured using regular SystemC (SC) and the first design (DV1), which does not use the new functions. It took 0.32 seconds. The next experiment achieved 0.75 seconds to simulate the first design (DV1) using SystemC modified for low power (SC-LP). The third and worst result was achieved when simulated using low power and using in design the new implemented functions (DV2).

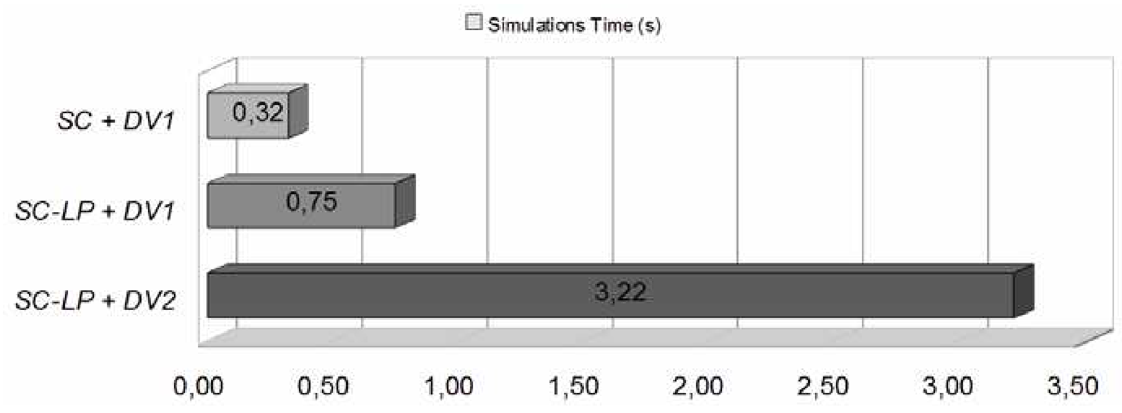


Fig. 19. Simulators performance

10. Simulator improvement

Due to simulator performance loss around 1000% compared with original SystemC, improvements were accomplished. This section presents that improvement to SystemC simulator with support for the functional verification of designs containing the principles of power gate design implemented in RTL. To demonstrate that the new modifications improved the performance of the simulator, the same techniques adopted in (Silveira et al., 2009) will be used.

10.1 Simulator optimization

The optimization of the simulator (Silveira et al., 2009) was accomplished based on the profiling of the running simulator, which demonstrated an excessive number of accesses to linked lists added to SystemC simulator kernel. A linked list is used to store the names of

the modules that must not be executed. The routine *sc_lp_turn_off* adds the module name to the list, while *sc_lp_turn_on* removes the module from the list, allowing it to be executed. Another list is kept to store the module constraints (wake delay). This list is required when the *sc_lp_add_constraint* function is called. In this case, constraints are added to the list and cannot be removed, only overwritten.
Based on profiling information, an asymptotic and semantic analysis of data structures used to implement the simulator kernel was performed. That consists of: (1) a new data structure to store information about which modules are turned off and the delay needed to retake full activity after its reactivation, (2) the data structure must provide information access at a very short and constant time interval.
The new functions were rewritten using a hash map to replace the linked list. Each hash map element represents a design module and is composed two variables (a boolean and a time). The boolean variable is responsible for identifying whether the module is activated or not, the time variable is responsible for storing the necessary time delay to re-activate the module. The elements are accessed using a key, which is the name of the module. The functions signatures have been altered, *sc_lp_add_constraint* was removed and its function was added to the routine *sc_lp_turn_on* and attributes are now passed to hash map. Table 3 shows how the functions signatures currently in *sc_simcontext.h* SystemC kernel file.

extern void sc_lp_turn_on (const char* module_name, sc_time wakedelay);
extern void sc_lp_turn_off (const char* module_name);

Table 3. Functions Declarations

10.2 Results

Among the simulations results, the preservation of the semantics and performance enhancement of the new simulator compared to the version shows in (Silveira et al., 2009) can be highlighted.
The improvement in simulator performance can be seen in Fig. 20. It can be seen that the design simulations (DV1) using the improved simulator (SC-LP-V2) presents an increasing of 4% in simulation time and simulations of power gate design (DV2) the increase of 8% in comparison with the original SystemC simulator.

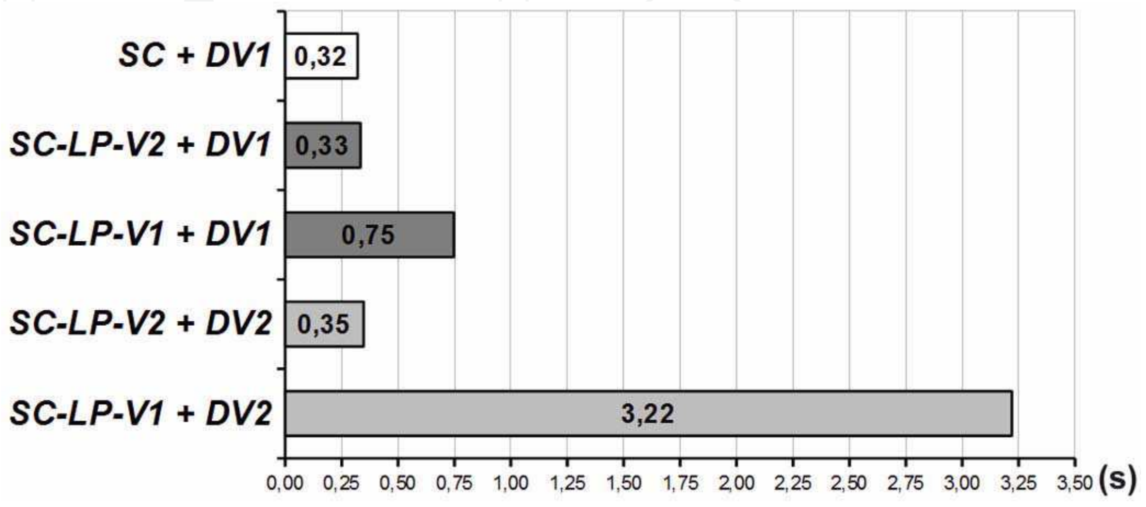


Fig. 20. Simulators performance

Comparing the two SC-LP simulators, the gains were significant. The SC-LP-V2 simulator achieved a performance increase of 224% in the execution of design without power gate design and 925% simulating power gate design. These performance gains were reached by eliminating the costs of elements addition and removal from linked lists and increasing the speed for accessing information through the use of hash map structure.

12. Final considerations

The innovative methodology presented here allows the modelling and simulation partially and dynamically reconfigurable hardware systems, enabling new functions to module blocking and resuming in the simulator kernel. This enables the dynamic behaviour to be foreseen before the synthesis on the target hardware (like FPGA). Furthermore, systems evaluation is possible even before their hardware description using a Hardware Description Language.

Even further, the same approach is being used to model and simulate low power hardware systems through power gate technique. The results prove that as dynamic reconfiguration, as low power systems can be simulated using the identical simulators. This opens new opportunities for both areas, enabling the tool exchanging for both proposes.

Our innovative methodology can be applied to any hardware simulator which uses an event scheduler. The main idea is to register each block that is not configured on a chip at a given moment during simulation. The simulator scheduler is programmed to not execute those blocked modules. We prove in this work that this approach covers every partial reconfigurable system situation. A particular strategy is also adopted to log the chip area usage enabling the investigation of the benefits of partial reconfigurations for each application.

13. References

Becker, J. & Hartenstein, R. (2003). Configware and morphware going mainstream. *Journal of Systems Architecture*. Vol. 49, No. 4-6, p. 127-142, September, 2003.

Becker, J., Vorbach, M. (2003). Architecture, Memory and Interface Technology Integration of an Industrial/Academic Configurable System-on-Chip (CSoC)", IEEE COMPUTER SOCIETY. ANNUAL Symposium ON VLSI, Tampa, Florida, February 20–21, 2003.

Becker, J.; Huebner, M. & Ullmann, M. (2003). Power Estimation and Power Measurement of Xilinx Virtex FPGAs: Trade-offs and Limitations". *Proceedings of the 16nd Annual Symposium on Integrated Circuits and System Design (SBCCI03)*, Sao Paulo, Brazil, September, 2003.

Brito, A. V.; Rosas, W. & Melcher, E. U. K. (2006). An open-source tool for simulation of partially reconfigurable systems using SystemC. *Proceedings of IEEE Computer Society Annual Symposium on VLSI (ISVLSI 2006)*, Karlsruhe, Germany, 2006.

Brito, A. V.; Kuehnle, M.; Huebner, M.; Becker, J. & Melcher, E. U. K. (2007). A General Purpose Partially Reconfigurable Processor Simulator (PReProS)" *Proceedings of 15th Reconfigurable Architecture Workshop (RAW'2007), 2007, Long Beach. 21st International Parallel & Distributed Processing Symposium*. Piscataway, New Jersey: IEEE, 2007.

Brito, A. V.; Kuehnle, M.; Huebner, M.; Becker, J. & Melcher, E. U. K. (2007). Modelling and Simulation of Dynamic and Partially Reconfigurable Systems using SystemC".

Proceedings of IEEE Computer Society Annual Symposium on Emerging VLSI Technologies and Architectures, (ISVLSI'2007), Porto Alegre. IEEE Computer Society Piscataway, Vol. 1. p.200 – 203. New Jersey: IEEE 2007

Dorairaj, N.; Shiflet, E. & Goosman, M. (2005), PlanAhead Software as a Platform for Partial Reconfiguration. *Xcell Journal*. Xilinx, Inc. December 2005.

Grotker, T.; Liao, S.; Martin, G. & Swan, S. (2002). System Design with SystemC. Kluwer Academic Publishers, 2002.

Huebner, M.; Becker, T. & Becker, J. (2004). Real-Time LUT-Based Network Topologies for Dynamic and Partial FPGA Self-Reconfiguration. *Proceedings of the 16nd Annual Symposium on Integrated Circuits and System Design (SBCCI03).* Recife, Brazil, September, 2004.

Keating, M.; Flynn, D.; Aitken, R. & Gibbons, A., Shi, K. (2007). Low Power Methodology Manual, For System-on-Chip Design, Series: Series on Integrated Circuits and Systems 2007, XVI, 304 p., Hardcover, ISBN: 978-0-387-71818-7

Keating, M.; Flynn, D.; Aitken, R.; Gibbons, A. & Shi, K. *Low Power Methodology Manual, For System-on-Chip Design*, Series: Series on Integrated Circuits and Systems 2007, XVI, 304 p., Hardcover, ISBN: 978-0-387-71818-7

Lysaght, P., Dunlop, J. (1993). Dynamic Reconfiguration of Field Programmable Gate Arrays. *Proceedings of the 1993 International Workshop on Field-Programmable Logic and Applications*. Oxford, England: Abingdom EE&CS Books, p. 82-94, 1993.

Pleis, M. A. & Ogami, K. Y. (2007). Dynamic reconfiguration interrupt system and method. *Cypress Semiconductor Corporation*, San Jose, CA, US. 2007.

Qu, Y.; Tiensyrja, K. & Masselos, K. (2004), System-Level Modeling of Dynamically Reconfigurable Co-Processors. *Proceedings of International Conference on Field Programmable Logic and Applications*, Antwerp, Belgium, August-September, 2004.

Qu, Y.; Tiensyrja, K. & Masselos, K. (2004). System-Level Modeling of Dynamically Reconfigurable Co-Processors", International Conference on Field Programmable Logic and Applications, Antwerp, Belgium, August-September 2004.

Rocha, A. K.; Lira, P., Ju, Y. Y., Barros, E.; Melcher, E. U. K. & Araujo, G. (2006). Silicon Validated, IP Cores Designed by The Brasil-IP Network". *Proceedings of IP/SOC Conference*, Grenoble, França, 2006.

Schallenberg, A.; Oppenheimer, F. & Nebel, W. (2004). Designing for Dynamic and Partially Reconfigurable FPGAs with SystemC and OSSS, *Proceedings of Forum on Specification and Design Languages (FDL '04),* Lille, France, September, 2004.

Schallenberg, A.; Oppenheimer, F. & Nebel, W. (2006). OSSS+R: Modelling and Simulating Self-Reconfigurable Systems. *Proceedings of the International Conference on Field Programmable Logic and Applications*, p. 177–182, August 2006.

Silva, K. R. G. & Melcher, E. U. K. (2005). A methodology aimed at better integration of functional verification and RTL design*, Design Automation for Embedded Systems*, Vol. 10, No. 4, p. 285-298.

Silveira, G. S.; Brito, A. V. & Melcher, E. U. (2009). Functional verification of power gate design in SystemC RTL. *Proceedings of the 22nd Annual Symposium on Integrated Circuits and System Design: Chip on the Dunes,* Natal, Brazil, August, 2009, SBC, Porto Alegre.

Zhang, X. & Ng, K. W. (2000). A review of high-level synthesis for dynamically reconfigurable FPGAs". *Microprocessors and Microsystems*, Vol. 24, No. 4, p. 199-211. August 2000.