Run-Time Partial Reconfiguration Simulation Framework Based on Dynamically Loadable Components

Xerach Peña(✉), Fernando Rincon, Julio Dondo, Julian Caba, and Juan Carlos Lopez

Universidad de Castilla-La Mancha, CP:13071 Ciudad Real, Spain
xerach.pena@uclm.es

Abstract. In this paper a SystemC-based framework for run-time partial reconfiguration modeling and simulation is introduced which allows to perform early design space exploration for dynamically reconfigurable systems. Besides, a middleware to extend the capability of TLM introducing a semantic to interconnect components described at different abstraction levels or languages is added. This middleware allows to automate the creation of the corresponding communication adapters to these new components. Finally, the services provided by the middleware are described and experimental results are presented to validate the proposal.

Keywords: SystemC · Dynamic reconfiguration · Middleware

1 Introduction

Nowadays, embedded systems composed by one or several microprocessors plus reconfigurable hardware are gaining importance in the implementation of a large range of applications. Their success is largely due to its flexibility, the capability to exploit hw reconfiguration and the high ratio of performance versus power consumption. In this context, partial reconfiguration capability is one of the most important features concerning many of these reconfigurable devices. However, although run-time reconfiguration enables new possibilities in the reconfigurable computation field, it also introduces new challenges such as how to get an efficient use of the reconfigurable resources, including how to obtain an optimum management of the reconfiguration process. These reconfigurable resources are used to host different components during design life-cycle. However, the characteristics of these resources such as occupied area size, shape, location, communication interface, etc., must be defined in the first stages of the design process, forcing developers to deal not only with traditional co-design techniques and high-level design problems, but with efficient reconfiguration process scheduling as well. Moreover, there is a lack of tools to handle the modeling of partially reconfigurable FPGAs at high level of abstraction making almost impossible to explore the impact of reconfigurable sub-systems on the performance and behaviour of the system as a whole in early design phases.

© Springer International Publishing Switzerland 2015
K. Sano et al. (Eds.): ARC 2015, LNCS 9040, pp. 153–164, 2015.
DOI: 10.1007/978-3-319-16214-0_13

To solve this problem, there have been some proposals, based on sw/hw systems for dynamic resource management in FPGAs, that combine both scheduling and instantiation tasks, and providing a complete flow management to support the design of dynamically reconfigurable hw [12]. However, this kind of solution lead designers to the need of a platform where designs could be tested and simulated, in order to avoid the implementation in hw until the design has been correctly checked out. Thus, the capability to model partial reconfiguration devices in this sort of platform is revealed as a fundamental requirement in the design flow. Furthermore, due to the ever increasing complexity of hw and hw/sw co-designs, developers strive for higher levels of abstraction in the early stages of the design flow.

In this context, SystemC [1] language has been revealed as an important tool to deal with not only modelling in high level of abstraction, but also to work at all stages of the design flow in hw/sw co-designs. SystemC allows the design and verification of sw, hw or mixed systems. It allows also describing a system at different levels of abstraction, from RTL up to functional models that may be timed or untimed. SystemC mainly consists of a class library of C++, composed of classes, macros and templates; and a simulation kernel, forming together a framework. This framework has the capability to model a concurrent system using modules, communications mechanisms and hw-oriented data types. A module is formed by one or several processes modeling its behaviour; ports to communicate with other modules; and internal variables to save its states. The communication ports can be interconnected by using channels. In a SystemC model hierarchy is allowed so a module can contain other modules. This hierarchy is dynamically constructed during the execution of the model elaboration phase and it cannot be changed once the simulation phase has started. This implies a major difficulty to model the partial dynamic reconfiguration using SystemC.

The simulation kernel executes a SystemC model scheduling and calling C++ functions registered by using SC_THREAD, SC_CTHREAD or SC_METHOD processes. This execution is divided into two phases: the elaboration phase and the simulation phase[2]. During the elaboration phase the modules are instantiated and initialized by executing their constructor. It is in this phase where the processes are also registered. Following, the connections between modules are established as a part of the initialization phase.

1.1 Dynamic Reconfiguration in SystemC

A reconfigurable system is one that can change part of its configuration at runtime. Dynamically reconfigurable FPGAs can modify part of its structure to implement different components with different functionalities in the same area whereas the rest of the system is running.

Modelling a reconfigurable system requires modelling its behavior as a whole, in where all possible configurations are taken into account. However, the main problem that a developer faces when trying to model dynamic reconfiguration in SystemC is that new instances can't be generated once the simulation phase has already started. This limitation can be solved by stopping the simulation

phase, opening the elaboration phase in order to make all necessary changes, and running again the simulation phase. However in this way the dynamic reconfiguration process is still not modeled.

Therefore, in a system formed by a set of microprocessor and dynamically reconfigurable FPGAs, a platform that allows the simulation, testing and verification of the system including dynamic reconfiguration at the first stage of the design, arises as a need.

To address these problems, in this paper a simulation framework is proposed, where reconfiguration takes place at two diffent levels. On one side, it allows the reconiguration of the interconnection infrastructure at run-time, where components can be described at different abstraction levels and languages, such as C/C++, SystemC RTL, or VHDL. On the other side partial run-time reconfiguration of components is modeled after the use of dynamically loadable libraries, which do not imply the modification of the simulation kernel, and provides a simple and homogeneous mecanism for the replacement of the different behaviors associated to a reconfigurable area. This framework is based on a lightweight middleware, using TLM-2.0 as the common transport layer.

The remainder of the paper is structured as follows: in Sect. 2, we discuss works related to SystemC and partial reconfiguration modeling. Section 3 introduces our simulation framework and the way to model dynamically reconfigurable systems, and Sect. 4 presents the application used to validate it. Finally, we discuss conclusions of our work in Sect. 5.

2 Related Works

Although there are some authors who have proposed a solution based on a modified SystemC kernel as presented in [3] and [4], most of the researchers have chosen to stay into the standard and not modify the simulation kernel.

Regarding an unmodified simulation kernel, works shown in [5] associated to ADRIATIC (Advanced Methodology for Designing Reconfigurable SoC and Application Targeted IP-entities in wireless Communications) Project, carries out the concept of Dynamic Reconfigurable Fabric. The fabric is a special component that contains several contexts and the capability to dynamically switch from one context to another. This paper propose a methodology for modeling dynamically re-configurable blocks, that takes as an input a SystemC model of a static system and transforms it into a code implementing a reconfigurable module. Candidate modules to a dynamic implementation are chosen by their common interface. The main drawback of this work is that it is necessary to have all the functionalities implemented in each reconfigurable module and it is not allowed to change it once the simulation has began.

Sharing the same point of view, authors in [6] present their approach to model partial reconfiguration as a SystemC library, called ReChannel. Although the concept of reconfigurable zone is added, a reconfigurable module consists in instantiating all possible modules for each reconfigurable zone, as the previous case, where only one module is activated at a time by a dynamic circuit switching.

To connect the set of reconfigurable modules and the remaining system, a portal is used by handling the basic channel defined on SystemC. The control to manage reconfiguration is carried out by dedicated portals. However, not only the tasks are assigned once at the beginning of the simulation, which means that it is not possible to move the task from one reconfigurable zone to another, but also, it is not allowed to add new modules in run-time either.

One approach closely related to the architecture of Xilinxs FPGAs, especially Virtex-II, Virtex-4 and Virtex-5 is shown in [7] and [8]. In this case, the authors put forward a simulation model where reconfiguration capabilities are included as well. The starting point is in the smallest unit that can be reconfigured at the hw level, generating a 1D sequence of columns or 2D grid of tiles, using a tile as a container encapsulating all needed functionalities. The inner process of a SystemC module is implemented using SC_CTHREAD, SC_THREAD and SC_METHOD to represent a synchronous sequential, asynchronous sequential, and combinational circuits, respectively. The method bodies only consists of a function pointer that indicates the currently configured functionality. Communication between tiles has been implemented by manifold ports, modelling the bus macros used as connectors in the FPGA. When components use more than one tile, there will be a master tile that execute the task while the others are functionally switched off, or simply route signals to neighbour tiles. This is an excellent solution to simulate dynamic systems in this type of FPGAs, however there are several aspects to take into account. Firstly, the set of ports per edge is given by the superset of ports required by all implemented functionalities. Thus every tiles must implement all possible communication protocols. Besides, every systemC module must include an empty function as well as those implementing simple signal transfers from incoming to outgoing ports, increasing simulation time. Finally, this simulation platform does not allow inclusion of new functionalities once the simulation has started, nor reuse it in architectures other than 1D or 2D meshes.

In [9], authors presented OSSS+R by adding reconfigurable support to the extension OSSS (Oldenburg System Synthesis Subset), and where they proposes the automatic synthesis of a reconfigurable system. This synthesisable subset of SystemC is extended by additional elements for high-level modelling, like shared variables, polymorphism or transaction level modelling. The C++ polymorphism concept is used to assign different modules to one reconfigurable zone, as long as they share the same static interface. This reconfigurable zone are modeled as a special container called osss_recon, which is the basis to take into account the reconfiguration and the context switch times to provide a RTL model. However, tasks are assigned to a reconfigurable zone within this container in a static way, and it is not allowed to change or add one in run-time.

The capability to enable, disable, resume and kill process on systemC were concepts introduced in SystemC-2.1 as dynamic threads, and it had been used by authors in [10]. In contrast of static threads, dynamic threads may be spawned during runtime and not only during the elaboration phase. In this work a reconfigurable module is composed of two dynamic threads: one for the actual user

process, modeling the functionality; and the other one for the creation and destruction processes (control). This work also define the concept of dynamic port, making possible the construction of a port after the elaboration phase. However the dynamic port comprises a set of static channels that generate instance of channel pool before the simulation, and manages the connection and disconnection of channels in the channel pool. Concerning the two dynamic threads that compose a reconfigurable module, the control thread is not part of the reconfiguration process, since it is not the module which is in charge of the reconfiguration process.

In [11] authors present a simulator, based on the separation of concerns between the application and the architecture, and it can be used either in early development stages, or during the implementation phase. As already mentioned in previous works, they use the concept of reconfigurable zone characterized by a set of resources. Through their methodology they have taken into account functionality changing and resources sharing. During simulation, a manager is used to map the reconfigurable modules into the appropriate zone according to available resources. In order to model the dynamic behaviour of the tasks, a module is set up with two dynamic threads. The first one is the User Algorithm and represents the functionality of a dynamically reconfigurable module, spawning at runtime. The second one is the Reconfiguration Control, that communicates with the configuration interface and it is responsible for the creation and the destruction of the User Algorithm. All the communications in this work (between modules and from the reconfiguration manager to a module) were described using OSCI TLM-2.0 standard.

Encapsulating a reconfigurable task, modeling its dynamic behaviour by SystemCs dynamic threads, and carrying out the communication by using TLM is by far the most flexible and complete method to work with partial dynamic reconfiguration on SystemC.

Extending some issues shown in related works, we presented in [12] a framework to manage partially dynamic reconfiguration in a completed transparent way for both the user and the application. In this work a Reconfiguration Engine has been designed to efficiently handle the reconfiguration process without processor intervention, and reducing considerably the reconfiguration time. Furthermore, we extended the use of TLM [13] not only to communicate between modules, or between a module and the remaining system, but also to show a method to widen flexibility concerning to the topology of the reconfigurable modules. Although the capability to change or add tasks in run-time on SystemC has not been successfully treated yet, in [14] is shown a method to work with plugins in C++ that opens a way for reconfigurable modules in run-time on SystemC.

Our approach relies on a combination between dynamic threads of SystemC and plugins of C++ to change the functionality of modules during run-time, and on transaction level modeling for all the communications. One of the most important features of our proposal is that it is not necessary to change the simulation kernel of SystemC, and therefore, it is in the line followed by the standard.

3 Simulation Framework

3.1 Heterogeneous Component Integration

One of the contributions of the paper is the definition of a common integration environment, where different kinds of heretogeneous (hw and sw) comoponents can be integrated at different abstraction levels. This is achived thanks to the use of a TLM communication infrastructure, and therefore to the separation of concerns between communcation and functionality. However, TLM is only used as the communication physical layer because, as novel contribution in this work, a middleware is added, incorporating a new logical layer, and appending a set of semantic rules to TLM (See Fig: 1).

The TLM layer works as a communication network that routes messages from one physical TLM destination to another. On the other side, each component has a logical identifier, and follows some simple and predefined rules for the construction and parsing of the communication messages, which define the fields to include in the message as well as the way data must be serialized to be consistent in both ends of the communication process. The purpose of the logical layer is to stablish a biunivocal correspondence between the behavior representation of the component as a set of functions and parameters, and how their invocations and return values are translated into messages. However, components may be modelled at any abstraction layer, as far as their final interface is compatible with the logical messages defined.

The separation of both the physical and logical planes of the communication is the key to provide transparency between components, and avoid unnecessary coupling between functionality and low-level integration details. However it relies on the use of a specific adapter, called CA (Communication Adapter), provided as part of the framework, and which will act as a bridge between both worlds.

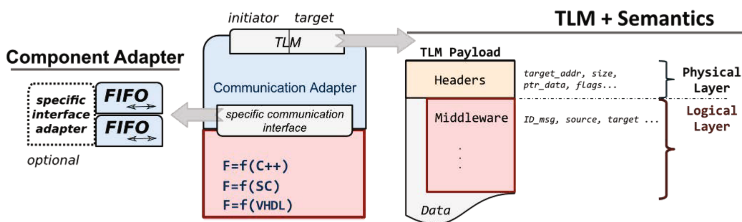

Fig. 1. Communication adapter

3.2 Communication Adapter Architecture

As a result, the simulation framework allows the interconnection of not only a set of components described in different abstraction levels, but also any other entity (such a external subsystem), as long as it is done through a specific CA. For instance, it is possible to connect our system to an emulator, where an specific

sw is running, and which makes use of a component simulated in our tool as a hw accelerator.

As its name describes, the purpose of the Communication Adapter (figure 1) is the adaptation of the component interface to the TLM communication infrastructure that links the rest of components in the system. On the TLM side, operation requests and data are encapsulated following the payload format described in the TLM-2.0 standard. On the component side, communication is performed by a double set of input/ouput FIFOs. The use of a FIFO interface may seem restrictive with respect to the use of a richer component interface at a first glance. However, the payload format used by TLM perfectly fits this approach with almost no overhead, and additionally, it is possible to add an specific adapter to map it into a more appropriate one.

The CA is a library component provided by the simulation framework. From the physical communication point of view, it can be configured to act as a TLM initiator and/or target. Each kind of interface has its counterpart input and output FIFOs in the component side. But from a logical point of view, the CA is the responsible for the implementation of the basic middleware services, which in the simplest case implies that communication transparency between components is guaranteed. This is achieved by the:

- **separation between physical and logical addressing spaces**. The physical one relates to the references used by TLM, which are used to route messages from initiators to targets and vice-versa. Logical references are used by the components, regardless of their physical location. Logical addresses are mapped into physical ones depending on the topology defined in the model. The mapping can be hardcoded in the CA, or may be delegated into a location service, such as the one described in section 3.6.
- **message format**. Messages are routed by the TLM infrastructure based on the information in their headers, which corresponds to the physical addressing space. The TLM data field of the payload can be divided into two parts. The first one can be considered the logical header of the message, and includes the logical address from the source and target, as well as a message identifier, that the CA can use to link return values with the requests. Finally the last part of the message includes the real data of the message, serialized following a set of fixed rules. That data can be directly forwarded to the component interface FIFOs.
- **message protocol**. In this case a simple request/replay protocol is used.

3.3 Partial Dynamic Run-Time Reconfiguration

As already mentioned in the introduction, SystemC suffers from some limitations when considering partial dynamic reconfiguration modelling.The main conclusion to be drawn from these limitations is that there is no direct way to add functionality to a component at runtime without modifying the simulation kernel.

Alternatively, this paper proposes of a novel technique for modelling partial dynamic reconfiguration using SystemC, which is based in the combination of

dynamic libraries, plus the use of a special component adaptor: the Reconfigurable Unit (RU). Such units can be configured to embed completely different behaviors, that may be replaced during the life cycle of the application. RUs and regular components are integrated into the simulation model using the CA described in section 3.2, since once configured, RUs act as any other component in the system.

RU reconfiguration is performed by the Reconfiguration Controller (RC). The task of the controller is to dynamically (at run-time) load the behaviors associated to each RU, and configure the CA to be accessible from other components in the system. The RU doesn't take any reconfiguration decisions by itself, but it provides a very simple abstraction layer for reconfiguration management. Currently the framework doesn't include any facility such as a reconfiguration scheduler, so it must be modelled as part of the application. However, nothing prevents from building it as a service on top of the RC.

Finally a directory service is required, in order to map logical references of the components into their physical TLM identity. As part of the reconfiguration process both the logical and physical identities of the component in the CA of the RU are updated.

3.4 Reconfigurable Unit Architecture

RUs are not real components but an almost empty frame that provides the FIFO interfaces described in section 3.2 (figure 2), as well as a reference variable that can instantiate different component behaviors, not statically determined (figure 2). The reconfiguration process then consists in the instantiation of a certain behavior included in a dynamic library, which can be loaded at runtime, just like a sw plugin. Once the library is downloaded, the reference is updated, and the RU behaves as specified in the library.

As it happens with plugins, something to take into account is that the interface of the component is fixed at design time, and therefore every component that maps into a RU must comply with it. However, that's not really a problem because in the proposed model, the entry point to the component are FIFO channels. If a more specific interface is required (for third party IP integration, for example), it can be easily adapted. The reconfigurable behavior thus will include the adaptor as part of the model. Furthermore, the existance of a fixed interface resembles the real physical behavior of reconfigurable logic in FPGAs.

3.5 Reconfiguration Controller

The Reconfiguration Controller includes a list of the available RUs in the system. This list is defined at compile, when the concrete number of available RUs is fixed. Therefore there is no limitation in the number of different behaviors used in the RUs, but the number of different RUs remains constant.

After elaboration, at run-time, the RC waits for the reception of reconfiguration requests. The RC works at a different plane than the RUs in the system, and does not share the same TLM communication infrastructure. The requests

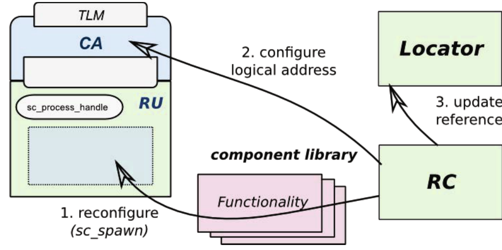

Fig. 2. Communication adapter

are received though the input FIFO as a serialized message, that can include one of the following type of operations:

- **start**: for the activation of the component, either because it has been previously stopped or it has just been configured
- **stop**: that disables both the execution and reception of messages
- **reconfigure**: the request includes the name of the behavior to download, as well as the physical and logical identities of the new resulting component.

From the simulator point of view, the reconfiguration process involves three operations, all of them managed from the RC:

1. First the binary code of the new behavior must be downloaded from a dynamic library. Previously the source code of the model must be compiled, and the compatibility of the interface has to be ensured. Once downloaded, the RU reference to the real code is updated (through a *sc_process_handle* reference).
2. The next step consists in the configuration of the logical addressing in the CA of the RU, so the new component becomes available to the rest of components in the system. At the same time such reference must be updated in the directory service (locator), so it can be queried by other CAs during logical to physical address translation.
3. Finally, a call to the *sc_spawn* function will initiate the execution of the process that acts as the entry point to the component behavior, which is equivalent to the *start* operation.

The procedure described above can also include time modelling, and a reconfigurable latency can be asociated in the form of a parametrizable function.

3.6 Location Service

This service is used to map physical and logical addresses, and can be easily implemented using a table. The use of a table even allows for the dynamic replacement of the references, a necessary feature to implement partial dynamic reconfiguration.

Components should only refer to the logical references of their targets so the models are not dependant from the concrete topology of the implementation. The translation between both of them should be a responsibility of the CA which will request for it to the location service. Such requests can be cached in the adapter to reduce the execution overhead.

4 Experimental Results

In order to check the validity of the approach two simple experiments have been carried out. The first one consists in the reconfiguration of a simple signal generator component with two versions, one producing a sinusoidal output, and the other generating a saw shaped signal. The top of the system simply instantiates one RU plus the corresponding CA, and the RC, and data communication reduces to a single message for starting/stopping the generation procedure. The result of the execution is shown in figure 3.

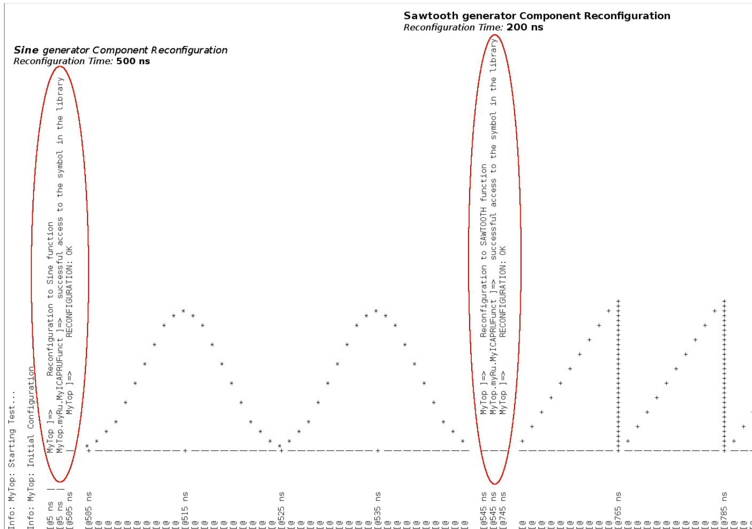

Fig. 3. Simulation of a reconfiguration

The second experiment consists in the simulation of a set of several versions of the Features from Accelerated Segment Test (FAST) Corner Detection algorithm [15] [16], used to detect corners in an image. In this experiment different versions of the algorithm are implemented in the same RU, and taking 9, 10, 11 or 12 pixels in the working out for each version. This experiment is used to model a dynamically reconfigurable part of a system in which the execution of this algorthm is accelerated in reconfigurable areas of a FPGA in which the hw implementation of the algorithm can be reconfigured in run-time according to

the version required [17]. In this experiment for each version, a golden reference model exists to check the correct performance of the algorithm. The top application uses the dynamically reconfigurable environment to keep the same system simulation environment to systematically evaluate all different versions. The model includes a test module that consecutively loads a different configuration into a single RU, injects the corresponding data through the TLM infrastructure, receives the results of the computation, and finally checks the correct performance of the model.

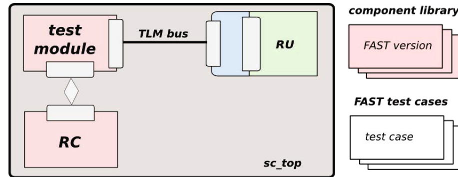

Fig. 4. FAST test simulation

However, the framework described is not limited to the use of a single dynamically reconfigurable component. Any number of them can be combined, including the static part of the design, or even linked with a third party models, as far as the communication takes place through the appropriate CA. This framework is specially well suited for the simulation of highly adaptable systems, such as the ones used in mixed mode genetic architectures, for example.

5 Conclusions

The main contribution presented in this paper has been a straightforward way to model partial dynamic reconfiguration in SystemC without the need to modify the simulation kernel of the language. Although the work mostly refers to reconfigurable components, it can be extensible to the whole model of the system, since the use of a communication middleware ensures communication flexibly and transparency between heterogeneous components. Such middleware is based in the use of TLM as the physical transport layer for messages, and therefore inherits the benefits in the modelling capabilities as well as the flexibility already provided by the standard.

Moreover, the proposed technique for reconfiguration implies almost no simulation overhead, since the number of components instantiated are kept to the minimum, there is only one active processes in execution per RU, and there is not any other in the background, disabled or waiting to be resumed. Finally, the extension of the system through the inclusion of new behaviors that can be instantiated into the RUs, implies no modification in the simulation model.

References

1. www.accelera.org
2. Black, D.C., Donovan, J., Bunton, B., Keist, A.: SystemC: From the Ground Up. Springer (2005)
3. Bhattacharyya, B., Rose, J., Swan, S.: Language Extensions to SystemC: Process Control Constructs. In: Procs. of the 44th Annual Conference on Design Automation (DAC), NY, USA, pp. 35–38 (2007)
4. Brito, A.V., Kuhnle, M., Hubner, M., Becker, J., Melcher, E.U.K.: Modelling and Simulation of Dynamic and Partially Reconfigurable Systems using SystemC. In: Procs. of ISVLSI, pp. 35–40 (2007)
5. Pelkonen, A., Masselos, K., Cupak, M.: System-Level Modeling of Dynamically Reconfigurable Hardware with SystemC. In: Procs. of Parallel and Distributed Processing Symposium (2003)
6. Raabe, A., Hartmann, P.A., Anlauf, J.K.: Rechannel: Describing and Simulating Reconfigurable Hardware in SystemC. ACM Transactions on Design Automation of Electronic Systems **13**(1), 1–18 (2008)
7. Albrecht, C. Pionteck, T., Koch, R., Maehle, E.: Modelling Tile-Based Run-Time Reconfigurable Systems Using SystemC. In: Proc. of the European Conference on Modelling and Simulation (ECMS), Prague (2007)
8. Pionteck, T., Albrecht, C., Koch, R., Brix, T., Maehle, E.: Design and Simulation of Runtime Reconfigurable Systems. In: Procs. of IEEE Workshop on Design and Diagnostics of Electronic Circuits and Systems (2008)
9. Schallenberg, A., Nebel, W., Herrholz, A., Hartmann, P.A., Oppenheimer, F.: OSSS+R: a framework for application level modelling and synthesis of reconfigurable systems. In: Procs. of the Conference on Design, Automation and Test in Europe (DATE), Belgium, pp. 970–975 (2009)
10. Asano, K., Kitamichi, J., Kenichi, K.: Dynamic Module Library for System Level Modeling and Simulation of Dynamically Reconfigurable Systems. Journal of Computers **3**, 55–62 (2008)
11. Duhem, F., Muller, F., Lorenzini, P.: Methodology for designing partially reconfigurable systems using transaction-level modeling. In: Procs. of Design and Architectures for Signal and Image Processing (DASIP), pp. 1–7 (2011)
12. Cervero, T., Dondo, J., Gomez, A., Rincon, F., Lopez, S., Peña, X., Sarmiento, R., Lopez, J.C.: A resource manager for dynamically reconfigurable FPGA-based embedded systems. In: Procs. of Conference on Digital Systems Design (EUROMICRO), Santander (2013)
13. Open SystemC Initiative (OSCI), OSCI TLM-2.0 Language Ref. Manual (2009)
14. Dynamic Class Loading for C++ on Linux. http://www.linuxjournal.com/article/3687
15. Rosten, E., Drummond, T.: Fusing Points and Lines for High Performance Tracking. In: Procs. of the IEEE Intl. Conference on Computer Vision (ICCV) (2005)
16. Rosten, E., Drummond, T.W.: Machine Learning for High-Speed Corner Detection. In: Leonardis, A., Bischof, H., Pinz, A. (eds.) ECCV 2006, Part I. LNCS, vol. 3951, pp. 430–443. Springer, Heidelberg (2006)
17. Dondo, J.D., Villanueva, F., Garcia, D., Gonzalez, C., Vallejo, D., Lopez, J.C.: Distributed FPGA-based Architecture to Support Indoor Localisation and Orientation Services. Journal of Network and Computer Application **45**, 181–190 (2014)