Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2009, Article ID 162416, 12 pages
doi:10.1155/2009/162416

Research Article

Multilevel Simulation of Heterogeneous Reconfigurable Platforms

Damien Picard[1,2] and Loic Lagadec[1,2]

[1] *Université Européenne de Bretagne, 35000 Rennes, France*
[2] *CNRS, UMR 3192 Lab-STICC, ISSTB, Université de Brest, 20 avenue Le Gorgeu, 29285 Brest, France*

Correspondence should be addressed to Damien Picard, damien.picard@univ-brest.fr

Received 17 December 2008; Accepted 15 April 2009

Recommended by Gilles Sassatelli

This paper presents a general system-level simulation and testing methodology for reconfigurable System-on-Chips, starting from behavioral specifications of system activities to multilevel simulations of accelerated tasks running on the reconfigurable circuit. The system is based on a common objectoriented environment that offers valuable debugging and probing facilities as well as integrated testing features. Our system brings these benefits to the hardware simulation, while enforcing validation through characterization tests and interoperability through on-demand mainstream tools connections. This framework has been partially developed in the scope of the EU Morpheus project and is used to validate our contribution to the spatial design task.

1. Introduction

Continuous advances in the VLSI processing technology enable to produce ever more complex systems on a single chip. These System-on-Chips contain dedicated circuits, processors, memories, and also reconfigurable circuits for increasing flexibility. As a result, an application executed on a RSoC is partitioned between the heterogeneous components of the system involving concurrent activities such as memory transfers, accelerator execution, and processor activities. The heterogeneity and the concurrent nature of the platform make it hard to program and to perform functional verifications of a running application.

In order to cope with the increasing complexity of SoC design, the abstraction level of the specification has been raised [1]. The VHDL and Verilog languages, the main hardware description languages employed today, support abstraction levels up to the functional level [2, 3]. However the lack of high-level programming features makes them unsuited for developing high-level models and systems.

The verification of an application running on an RSoC has to take into account all the system activities and their impact in term of behavior and performances. For example, an application mapped on a reconfigurable accelerator is strongly dependent of the communications, managed by a DMA controller, between the local buffers and the main memory. Therefore, the execution of an application can be considered as concurrent and involves specific issues for testing and debugging such as collecting local states in order to build up back a snapshot of the whole execution state.

A higher abstraction level entry for SoC design is achieved by extending mainstream programming languages such as C/C++ or Java [4–6]. The well-known SystemC enables to describe and simulate hardware/software at system-level by using a C++ class library [7]. The particularity of the SystemC approach is that it brings to SoC design the advantages of object-oriented software development. Some other works using object-oriented concepts in various languages have been reported such as [8–11].

However, the mainstream environments used (e.g., SystemC) have restricted debugging exploration capabilities and lack of dynamic features such as hot-code replacement or object state alteration at runtime. Furthermore, despite they are oriented toward software development, engineering methodologies such as eXtreme programming remain underexploited keeping mitigated the productivity gain.

This paper focuses on the simulation, testing, and debugging of an application running on an RSoC. The

presented methodology aims at bringing to RSoC application design the agility and efficiency of software development techniques as found in object-oriented languages and eXtreme Programming (XP) methodologies [12]. The proposed framework enables to model the platform at a system level by assembling components that inherit from an object framework. The application mapped on a reconfigurable accelerator is specified at different abstraction levels and integrated as a component in the system model. The specification starts at the behavioral level as object-oriented code which is refined automatically through an intermediate representation (IR). This IR is a Control/Data Flow Graph (CDFG) encoding algorithms and is taken as input of simulators and synthesis tools [13].

The framework is developed in a Smalltalk-80 object-oriented environment which offers symbolic debugging facilities and testing features as well as tools for exploring the application at runtime giving access to instantaneous snapshots of the system state [14, 15]. Our framework brings these benefits to the hardware simulation, while enforcing validation through characterization tests. Our approach relies on a nonmainstream environment, therefore, interoperability with mainstream tools is a critical issue and is provided through automated code generation.

The article is organized as follow. Section 2 discusses software engineering methods and tools support for validation and how their concepts are applied to our framework. Section 3 details the intermediate representation of the applications supporting multilevel simulation and used for synthesis. The system-modeling framework based on components is described as well. Section 4 presents the simulation models used by the framework and the automated generation of HDL wrapper enabling interoperability with mainstream tools. Section 5 shows simulation results at multilevel for the system and the mapped applications.

2. Software Tools Support and Methods for Validation

Software engineering methodologies are well known for enhancing productivity. This section presents tools and methods used in software design for validating applications and how it is applied to our methodology.

2.1. Debugging Tools and Use. Debugging is the task of tracking down causes of program malfunction. Some subtle errors, such as the mishandling of unusual assignments to a variable, can take a lot of exploration to trace and resolve. To trace these the programmer needs a mechanism for tracing the flow of a program and variable assignments at various points.

VisualWorks Smalltalk [15] provides several facilities to help the programmer debug his programs. Software probes insert triggers into the compiled code, without changing the source code, which interrupt either processing (breakpoints) or log status information (watchpoints). A walkback window is opened when an unhandled exception is detected, showing the last several executed methods. The debugger tool allows for extensive exploration of the execution history, for modifying variable values, and modifying code on the fly, and for controlling program execution.

```
test1
    self shouldnt: [CDFGSynthesisAPI example1]
        raise: TestResult error
```

Algorithm 1: Validating an example. The method test1 is defined in a class inheriting from the SUnit framework. It provides a set of methods that detect all conditions that return false while handling unexpected errors.

To diagnose a problem, sometimes it is sufficient to see the last few entries in the context stack. The Debugger's top view lists as much of the stack as the programmer wants to see. Then, one would like to continue execution in the debugger for the next iteration for some iterator construct. The software debugger provides these capabilities.

Software probes provide a way to check the state of the system at a specific point. A software probe does not change the source code design but will affect the timing of the program execution. Similarly, using an electronic probe does not change the design of an electronic circuit, but, when used, it may change the circuit's characteristics slightly.

A breakpoint, which is the simplest kind of probe, immediately opens the system debugger, skipping the notifier stage, when it is triggered. The top method in the stack is the method containing the breakpoint.

A conditional expression may be used with a breakpoint, allowing the programmer to test for specific conditions and selectively trigger the breakpoint. The expression, that must return a Boolean upon completion, can include any arbitrary operation such as data collection.

Debugging makes an intensive use of such mechanism as gaining an in-depth understanding of a program behavior often goes through a cyclic speculate-test scheme. Stressing a hypothesis then relies on probes usage.

2.2. Methods: Iterative and Test-Driven Development. Developing reusable software typically involves many design iterations. Each iteration may introduce new requirements that change or extend the original design. This iterative process of rearchitecting a design may be described as code refactoring. Whereas reworking or rewriting code may involve dramatic changes in functionality; refactoring is an intermediate step that generally does not disturb the behavior of an application.

Extreme Programming [12] advertises the creation of unit tests for test-driven development. The developer writes a unit test that exposes either a software requirement or a defect. This test will fail because either the requirement is not implemented yet or it intentionally exposes a defect in the existing code. Then, the developer writes the simplest code to make the test, along with other tests, passes (Algorithm 1).

XP mandates a *test everything that can possibly break* strategy and provides guidance on how to effectively focus the limited resources we can afford to expend on the problem.

```
test2
    CDFGSynthesisAPI new
                example: 'example.step'
                family: 'F5'
          resF5 := self readResult.
       CDFGSynthesisAPI new
                example: 'example.step'
                family: 'F4'
          resF4 := self readResult.
     self assert: resF5 = resF4
```

Algorithm 2: Characterization-based validation for synthesis cases on two reconfigurable device families (F5 and F4). Both postsimulation memories contents are checked to be equal.

XP's thorough unit testing allows several benefits, such as simpler and more confident code development and refactoring, simplified code integration, accurate documentation, and more modular designs. Besides, these unit tests are also constantly and automatically run as a form of regression test.

The main drawback using such tests in a simulation scope is obviously to write significant tests, whereas there may be no specification available for the process being simulated.

Characterization tests act as a replacement. A characterization test characterizes the actual behavior of an existing piece of software and therefore protects existing behavior of legacy code against unintended changes via automated testing [16].

The goal of characterization tests is to help developers to verify that the modifications made to a reference version of a software system did not modify its behavior in unwanted or undesirable ways. They enable, and provide a safety net for, extending and refactoring code that does not have adequate unit tests.

2.3. Moving from Software to Hardware Design. Despite hardware debugging is a major concern, getting such facilities as speculate-test pairing support and deep state analysis are not obvious to offer in a simulation framework. One solution relies on some design patterns such as *memento* and *observer* [17]. Respectively, they are used for state recovering and propagating changes through dependent objects.

Evaluating conditions before clocking the circuit when running simulation allows to freeze the execution on demand, hence implements conditional probes. Recording a stack of states allows to trace some bugs back from a conditional break point.

In addition, the simulation framework provides feedback when refactoring the CDFG that models the system and offers by then an iterative development support. For example, the designer may change the I/O data types or substitute a parallel node by a sequential one. Further, as some portions of the initial CDFG are replaced by RTL ones during the synthesis process, refactoring also covers migrating from high-level to low-level CDFG while preserving the agile behavior of software development. As a low-level CDFG is an extension and so a special case of high-level CDFG, synthesis appears as a deep refactoring similar to source-to-source transformation at software engineering level.

As our simulation framework operates at several abstraction levels ranging from untyped functional code (Smalltalk) to RTL netlists, a set of characterization tests can be automatically generated to validate the synthesis scheme. This is done by exercising the one step above specification level (code, CDFG) with a wide range of relevant and/or random input values, recording the output values (or state changes) and generating a set of characterization tests. When the generated tests are executed against a new version of the code (e.g., synthesized CDFG), they will produce one or more failures/warnings if that version of the code has been modified in a way that changes a previously established behavior. Characterization tests remain change detectors; it is up to the person analyzing the results to determine if the detected change was expected and/or desirable, or unexpected and/or undesirable.

Algorithm 2 illustrates a characterization test for validating synthesis cases on two reconfigurable device families (F5 and F4). Both produced CDFGs are simulated on a common set of inputs. The output values are stored into memories. Identity between these two memory contents is assumed to point out that both CDFGs share a common behavior.

3. Application and System Modeling

Our multilevel simulator takes as input a specification of an accelerated application and a model of the host system. The global flow of the methodology is depicted by Figure 1.

At the highest level the application is specified by behavioral code. Entry point is not restricted to a particular language since simulation can be performed on an intermediate representation (IR), which is a Control/Data Flow Graph (CDFG), generated from the behavioral code. However, to simulate the application at a behavioral level the language has to be the same as the framework since application is executed as software.

The high-level CDFG is synthesized to a lower level, by our tool named Madeo+, and can be simulated or exported to EDIF as well as Verilog netlists. Target dependent tools perform final place and route.

Simulation of the application is integrated in a system-level simulation for a global simulation. The host system is modeled from an object framework defining components and communication links. The multilevel simulator produces Gantt and interaction diagrams giving information on the system behavior as well as signals waveforms generated by the application at RTL level.

Debugging and testing iterations are performed on simulation and synthesis results.

3.1. Multilevel Representation Supporting Simulation

3.1.1. Intermediate Representation for Modeling and Simulating Applications. A CDFG is a directed graph whose nodes are operations and edges represent data flows. A classical representation of the CDFG consists in isolating data

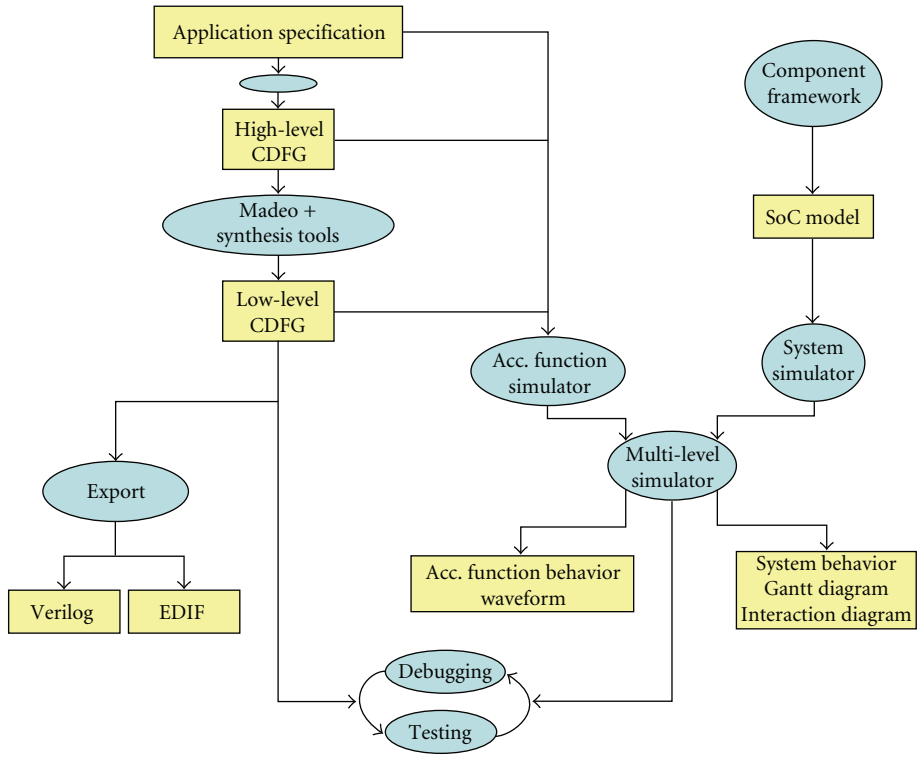

Figure 1: Global flow.

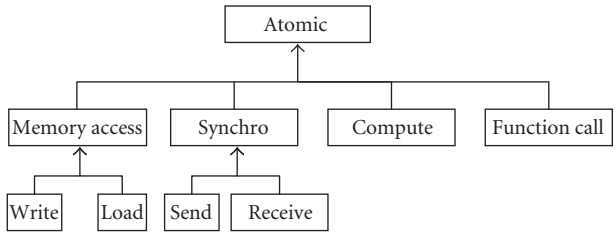

Figure 2: Atomic node inheritance tree in our CDFG model.

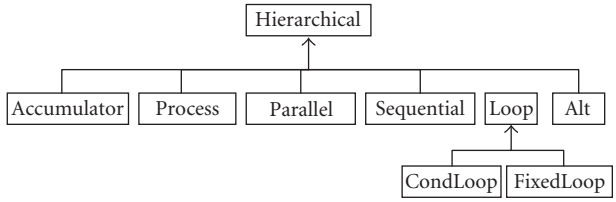

Figure 3: Hierarchical node inheritance tree in our CDFG model.

nodes constituting a DFG apart from control nodes (CFG) which role is to drive the execution flow of operations by establishing a specific instruction ordering (e.g., loops and conditions). Representing a CDFG requires to manipulate nodes and edges. Edges represent data; a datum classically knows its source, its consumers, and its type. An edge cannot cross a hierarchy; to connect to a node located into another hierarchy, the datum must be encapsulated within an *AliasData* which allows (1) a better modularity of the hierarchical CDFG and (2) renaming of signals.

CDFG nodes carry their own semantic. They do not explicitly dissociate the control from the operative nature. For example, a loop is a particular hierarchical node; it is not a control structure in charge of the execution of the loop core.

The CDFG is defined in an object-oriented environment; Figures 2 and 3 give the class hierarchy of the model for both types: atomic and hierarchical.

Atomic Nodes. An atomic node (see Figure 2) is a node without suboperators; it represents an abstract hardware primitive. If hardware primitives are not available in the target or in libraries, nodes are implemented as soft macros.

A computation is modeled by a *ComputeNode*, which result is returned as the output edge of the node. However, to simplify the detection and the processing of specific functions, some particular compute operations are handled as specific operators: memory accesses (*LoadNode*, *WriteNode*), communication, and synchronization on channels in a CSP like formalism [18] (*ReceiveNode*, *SendNode*).

Hierarchical Nodes. A hierarchical node (see Figure 3) is a node containing suboperators. We can distinguish three types of hierarchical nodes.

(i) Concurrency: *ProcessNode* for implementing parallel coarse grain operation (*threads*), *ParallelNode*, for parallel fine grain operations, or *SequentialNode* for sequential ones.

(ii) Control: loops (*FixedLoopNode* and *CondLoopNode*), alternative (*TestNode*, *AltNode*), and accumulation (*AccumulatorNode*).

(iii) Semantic-less organization: *CompositeNode* that can come from loop bodies or functional calls.

3.1.2. RTL Modeling

(a) Elementary Components. The circuit appears as a set of operators and data; the data carry their source and consumers, and the operators keep a link to their data's IOs. These double-linked dependencies enable to simulate the circuit.

(b) Object-Oriented Modeling: Polymorphism and Synthesis. The RTL modeling makes use of another structure, qualified as low-level CDFG, partially inherited from the CDFG one, but that is linked to the target platform. This structure conforms to the abstract node/data schemata (defined in the EXPRESS formalism [19] and generated using Platypus tools [20]). As this RTL model complies to the CDFG's application programming interface, and due to object-oriented facilities, these two levels can be interleaved within a single model to be simulated.

This model includes additional constructs (soft macros), primitive operators (libraries), registers/flip-flops as well as random logic nodes (e.g., BLIF format), and FSM description (e.g., KISS format). This model supports outputing EDIF files; hence it allows coupling with back-end tools providing among other feedbacks the circuit operating frequency.

Pushing forward the extension of the high-level CDFG to build up a low-level CDFG is a natural way to tailor RTL modeling. This ensures that a low-level CDFG can easily fit to any target at the cost of redefining the library of primitive operators. Such an operator is specified by its I/O bitwidth and its behavior to permit mapping from high-level CDFG nodes as well as by its latency which is used for timing generator during synthesis. An alternative solution consists in relying on random logic to design operators.

3.2. Object-Oriented System Modeling

3.2.1. A Component Approach for Structuring the System. SmallSystem is an object-oriented framework providing to the designer a set of abstract entities for modeling concurrent systems, built as a set of components interconnected through communication ports. Concepts implemented in SmallSystem are independent from any implementation language and could be retargeted to another environment/language. However, our approach takes advantage of a Smalltalk-80 environment that provides dynamic features as mentioned above [15].

An object-oriented framework provides to the designer a set of software components that can be directly used or tailored for a given application by subclassing. An application appears as an extension of the framework. The more it is subclassed, the more it is enriched with new reusable functionalities, bringing to the designer a flexible resource. Moreover, all the classes of a framework inherit from common classes' hierarchy. Therefore, they share a set of methods allowing to apply generic algorithms to all the elements of the framework. The tools addressing the framework are developed independently from specific extensions guaranteeing a high-degree of reusability.

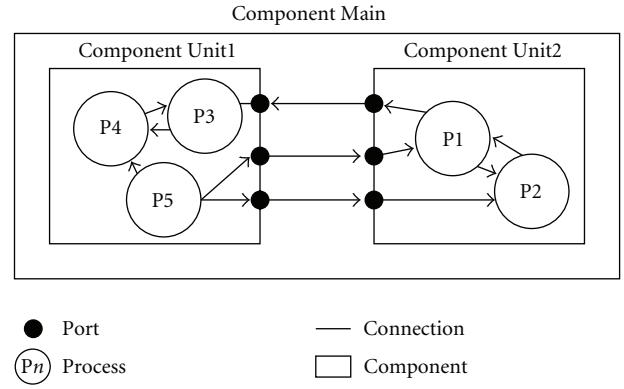

Figure 4: Example of a model with three components. The top hierarchy is the component *Main* that encapsulates two subcomponents *Unit 1* and *Unit 2*. Internal activities of each subcomponent communicate through the ports of the interfaces.

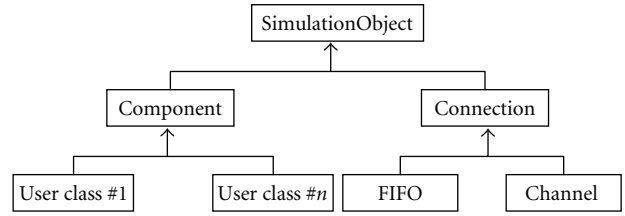

Figure 5: Class hierarchy of the modeling framework SmallSystem. In order to use simulation functionalities the framework SmallSystem inherits from the simulator.

SmallSystem defines abstract entities that are specialized by inheritance for modeling concrete system elements and communication links. An abstract system element is structured as a component implementing a behavior and a communication interface defined as a set of input and output ports. A system is built by interconnecting the component's interface with communication links of varying semantics and capacities (see Section 3.2.2). A component encapsulates its internal states and behaviors meaning that only communications with its interface can modify them. For integrating a component in a system, only its functionality and interface have to be known. This approach enables to define concrete elements that are structurally independent from a specific model. Additionally a component can be hierarchical; it then contains an entire subsystem being connected through the interface of its encapsulating hierarchy to an external component. The internal activities of an encapsulating component can also communicate with a subcomponent.

Figure 4 illustrates an example of a hierarchical model. The top hierarchy is *Main* encapsulating the two components

Unit1 and *Unit2*. *Unit1* defines its internal activities by three processes. Its interface defines three ports, and two processes are in charge of the external communications. Internal communications between processes are performed by local declarations of communication links.

3.2.2. Raising Up the Abstraction Level: Object-Oriented Modeling. Figure 5 illustrates the class hierarchy of the SmallSystem framework. The classes *User Class (1 and n)* correspond to an extension of the framework.

The basic framework elements are as follows.

(i) Component. It represents abstract behavioral system elements. A specialization adds behaviors to the component by the definition of additional methods corresponding to internal activities. These methods can be executed as processes or used as classical functions. An interface is defined by the declaration of input/output ports in an initialization method. The internal states are represented by instance variables and are not directly accessible by external components.

In order to access the simulation features this class inherits from the simulator framework. More details about the simulator are given in Section 4.3.1.

(ii) Connection. This abstract class corresponds to a connection between two component ports and allows to perform communications. At this abstraction level no semantic is associated to the connection. The class is specialized in two entities that are Channel and FIFO. The former is an equivalent of a CSP channel as defined in Occam language [21]. Communications are synchronized by *rendezvous* providing a fine-grain synchronization. The latter is a classical FIFO with a parametric size. Communications are blocking if either the FIFO is empty or the maximal capacity is reached (this parameter is defined by the designer).

Because of the delays created by the synchronizations when the model is simulated, this class also inherits from the simulator framework.

4. Methodology of Simulation

A multilevel simulation needs to manage different models addressing the different abstraction levels. This section details the simulation models used by our framework and how they interact. The simulation engines for system-level and RTL-level simulation engines are described. Automated HDL wrapper generation ensuring interoperability of our framework with mainstream tools is explained.

4.1. Simulation Models. A simulation model defines the evolution of the system's states according to the simulated time. The model we use for system-level simulation is based on discrete states with an event-driven approach while the RTL level circuit simulation is cycle-accurate, that is, time-driven.

An *event* corresponds to a change of state in the simulated system caused by incoming data or by internal processes. A system state simulated in a discrete event model is defined by discrete state variables. Events are produced at discrete instants called *dates*. Continuous variable values, including time, are taken into account only when they are used, that is, at event dates.

Discrete event simulation can be classified according to how the simulation time goes on: event-driven and time-driven.

Event-driven model does not use global clock; the time is updated with the next earliest event date. Thus the time evolution only corresponds to an ascending order of the dates. The event scheduler consumes the event queue in this order, triggers the events, and updates the simulated time. Because of the sparse event repartition in a system simulation, the event-driven model is well suited for behavioral system simulation compared to RTL level. It avoids simulating empty cycles.

The simulated time of a time-driven model is incremented by a constant time step that is set according to the problem. For each time step, the event scheduler searches for dates in the list of events corresponding to the current step, then the selected ones are executed. Obviously, it is possible to obtain a time step with no events scheduled wasting simulation time. Time-driven simulation is interesting for an RTL level circuit simulation because of the activities produced at each clock cycle such as updating values in flip-flops. In this case the time step is set to one clock cycle.

4.2. Embedding Cycle-Accurate Simulation within an Event-Driven Scope. The platform described in this paper combines two complementary simulation models for simulating an application running on an RSoC. The system level is simulated by a discrete-event and event-driven simulator whereas the RTL level, corresponding to an application mapped on an accelerator, is simulated by a cycle-accurate simulator. This configuration raises the integration problem of both simulation models with separate notions of time. In order to synchronize both models, the cycle-accurate simulation is embedded within the discrete-event one as an atomic task. At the discrete-event level, when a start signal is received, for example, from a DMA controller, the mapped application is executed by starting the cycle-accurate simulator (for more details on the API used to interface both simulators see Section 4.3.2(b)). When computation ends, it returns the amount of cycles spent. This latency is converted in the time unit used by the discrete-event simulator in function of the accelerator frequency. The result corresponds to the delay applied in the event-driven simulation for simulating the application execution. Although the mapped application is time-driven, at the system-level the simulation is homogeneously event-driven.

4.3. Simulation Engines

4.3.1. System-Level Simulator. The high-level discrete-event simulator is event-driven and mainly inspired by [14]. The simulation kernel is based on a scheduler providing all the functionalities related to the creation, the scheduling, and the execution of events. A simulation model is composed of simulation objects that create delays for simulating execution

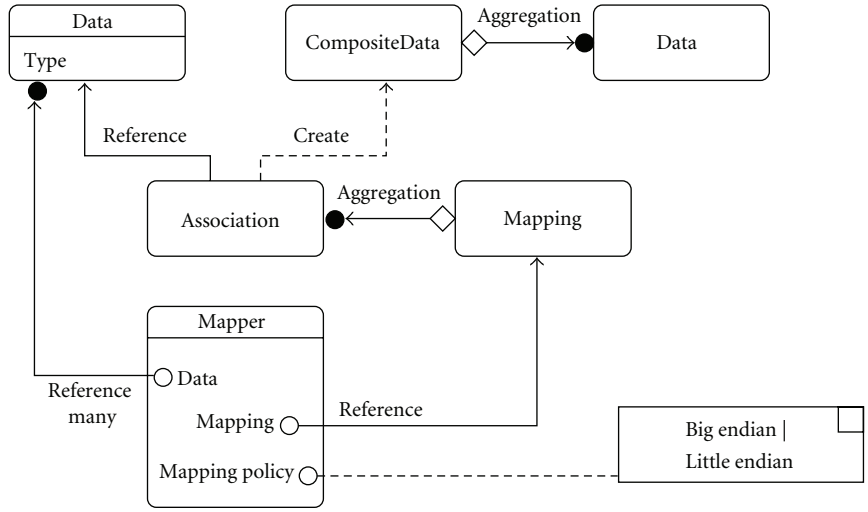

Figure 6: The mapper links high-level data to an aggregation of Boolean data, with respect to data type.

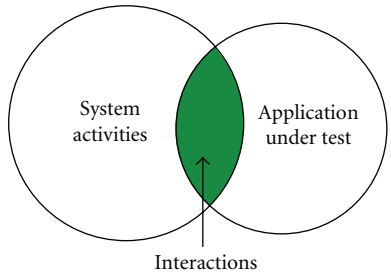

Figure 7: Interactions between the design under test and the host system correspond to the activities emulated by the mock object.

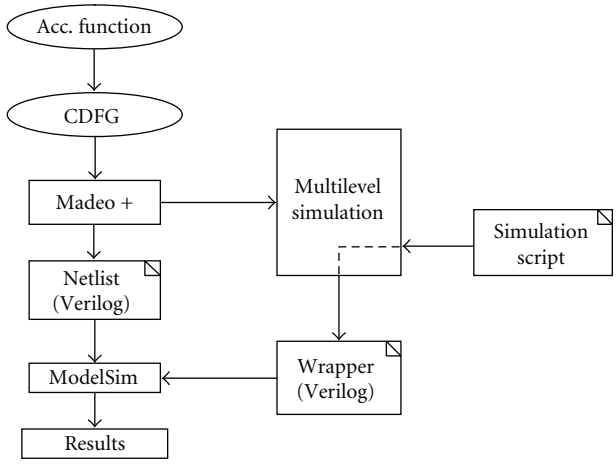

Figure 8: Simulation with generated wrapper and third party tool: ModelSim.

latencies and schedule activities. The initial scheduling of Smalltalk methods belonging to the components performs the simulation of a system model. They represent the component internal activities and are executed as processes.

Methods are executed by the Smalltalk virtual machine and by default spend no time in the simulation. Latencies of tasks are simulated by the insertion of specific calls to the simulation kernel features for creating delays and tracing the activities. The delay specified can be constant or generated in function of a probability law for simulating nondetertiministic behaviors such as bus contentions.

This technique allows monitoring fine grain system activities and allows multiple levels of detail. For example a task can be seen as atomic or traced at different points for more accuracy. The simulation trace obtained is used by the visualization tools presented in Section 5.1.

In order to have access to the simulation kernel functionalities, the modeling framework inherits from the simulator class *Simulation Object* as illustrated by Figure 5.

The simulation kernel defines two abstract classes.

(i) Simulation Object. This class corresponds to an abstract simulation object and gives a restricted access to the simulation kernel. It only provides to its subclass a set of methods for scheduling and suspending activities. For example, a simulation object has no direct access to the event queue under the responsibility of the scheduler. The components and communication links of the modeling framework inherit from this class.

(ii) Simulation. This class is the entry point of a simulation model. It permits to initialize the simulator and to start a simulation by scheduling the initial activities.

4.3.2. Cycle-Accurate Simulator for Mapped Applications. The simulation consists in computing the current value for the data. Simulation is performed at RTL level and is cycle-accurate.

Simulation relies on the following circuit structure.

(i) *Data* are computed by operators.

(ii) *Constant Data* force evaluating their consumers.

(iii) *Operators* compute their outputs following a data driven scheme.

(iv) *Flip flops* own a special behavior to delay their change until the next clock edge.

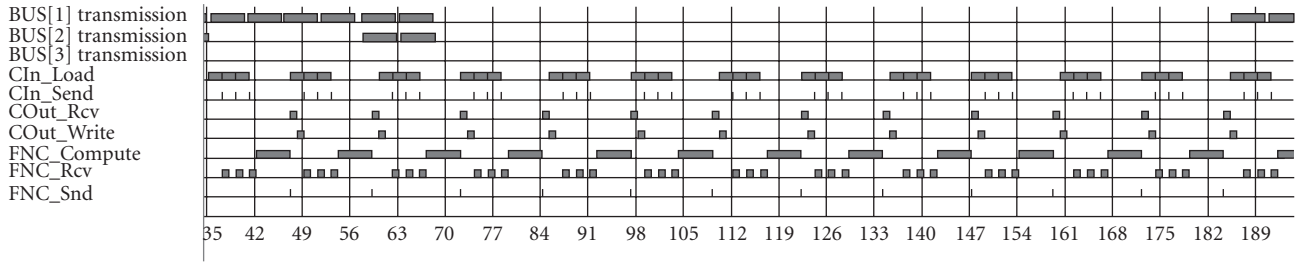

Figure 9: Gantt diagram generated by the system simulator.

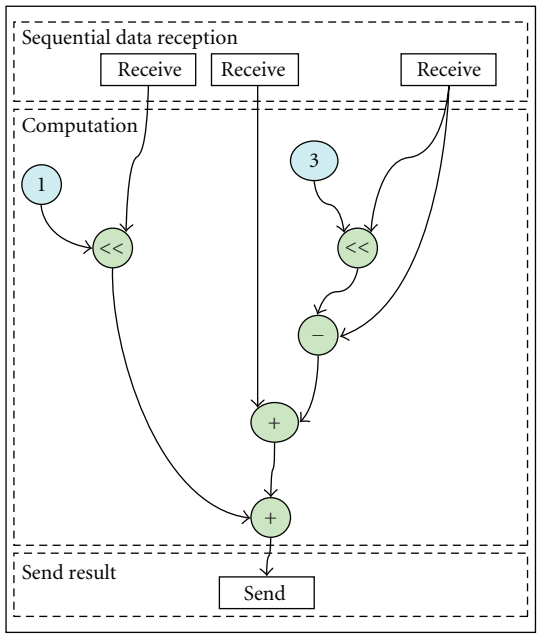

Figure 10: Application is an FIR filter function connected to input and output address generators (not represented on this figure).

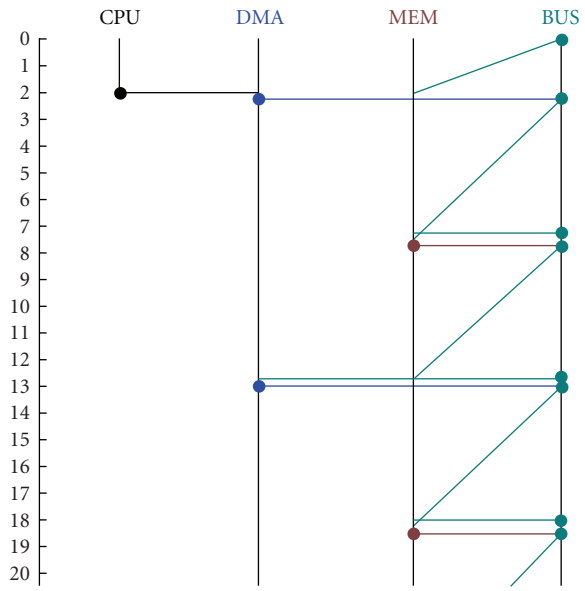

Figure 11: Interaction diagram generated by the system simulator.

The algorithm iterates over data and tries to fire their consumers evaluation. *Evaluation* is correct when all data changes have been considered.

This algorithm cools down until data values are stabilized. Clocking the circuit validates the flipflops' values and evaluates again.

(a) High-Level CDFG. As the mapped structure keeps alive the link between high-level variables and low-level signal (see Figure 6), both low-level and high-level values can be simultaneously probed during simulation (the *mapping Policy* is used to rebuild the high-level values based on the signals Boolean values).

(b) Low-level CDFG Simulator API. The low-level simulator provides an API enabling to set conditional breakpoints on the low-level CDFG top interface signals. SmallSystem accelerator components interact with the low-level CDFG under simulation through the API by defining execution scenarios in a method.

(c) Probing CDFGs. Two kinds of conditional breakpoint are used which either stops the simulation or performs an action when triggered. Actions and breakpoints are defined in a metamodel which is instantiated and used by the simulator. This model can also be used for generating an HDL wrapper for simulation with third party tools (see Section 4.4).

Algorithm 3 gives the syntax for a conditional breakpoint set on the application termination signal stopping the simulation when signal *done* is equal to 1. It corresponds to the instantiation of a probe object with an implicit action. The signal is an output and is retrieved by its name in the low-level CDFG through the simulator API.

In order to simulate response on signals, it is also possible to define breakpoints that perform user-defined action. Algorithm 4 defines a conditional breakpoint associated to an action setting the signal *dma_Req_Read* to value 0 after 10 cycles when signal dma_Ack_Read is equal to 1. The message *forceSignal:* sent to the class *SimulatorForceSignal* creates an action executed by the low-level simulator.

4.4. Automated Wrapper Generation for Third-Party Simulation. Stimuli are performed by a wrapper mimic behavior of the external environment such as systems activities. In software engineering and object-oriented development this technique is referred as *mock object*. A mock object simulates a real object by defining the same interface and

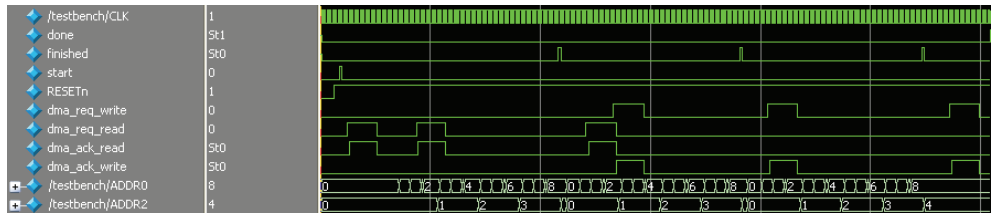

Figure 12: ModelSim results.

```
Simulator
    probe: (Simulator synthesizedCDFG
                                outputNamed: 'done')
    relation: '='
    value: 1
    probeName: 'Task Done.'
```

Algorithm 3: Conditional breakpoint set on the application termination signal. When the breakpoint is triggered, the simulation is stopped.

```
Simulator
    probe: dma_Ack_Read
    relation: '='
    value: 1
    probeName: 'dma_ack_read'
    action: (SimulatorForceSignal
                        forceSignal: (dma_Req_Read)
                        to: 0
                        in: 10).
```

Algorithm 4: Conditional breakpoint associated to a user-defined action. The signal dma_Req_Read is set to 0 when the breakpoint is triggered. The action can also be conditional, and multiple actions can be defined in an array.

providing equivalent services but without implementing the real functionalities. It enables to quickly validate the behavior of another object in a controlled way.

In a standard hardware design flow, applications are validated through HDL simulations performed by mainstream tools such as ModelSim [22]. Validation goes through using a wrapper written in HDL which defines a set of stimuli interacting with the application's top interface. Writing HDL wrappers can be burdensome and time-consuming when applications have complex interfaces. Furthermore each wrapper is specific to a given application, and it is necessary to rewrite it for addressing different cases. In order to ensure the interoperability of our methodology with mainstream tool flows and to increase productivity, HDL wrappers are generated from a higher-level specification.

In our framework, system activities, such as data transfers, are modeled by SmallSystem (see Section 3.2) and interfaced with the application through the low-level simulator API. However, it is also possible to test the application apart from a system model by defining stimuli in a standalone script. Scripting produces stimuli by instantiating the metamodel described in Section 4.3.2(c); stimuli correspond to breakpoints with actions simulating system activities (normally handled by the system model, e.g., memory requests) that interact with the design's top interface. The mock object corresponds to the probing model made up of the set of stimuli.

Activities emulated by the mock object correspond to the intersection between system activities and the design (see Figure 7), for example, DMA handshakes and memory requests.

This software approach takes advantage of testing and debugging features as described in Section 2. The abstraction level of scripting enables to be more productive; moreover, low-level aspects are taken into account by the simulator (scripting and generating HDL wrapper enables to save 50% of the designer's coding effort compared to hand-written HDL wrapper). For example, signal declarations or design instantiation is not necessary since the simulator gives access to the top interface through an API.

In order to ensure the interoperability of our methodology and final validation by mainstream tools, a Verilog wrapper is generated from the mock object. This corresponds to a model-to-model transformation with refinement on data. A probe on a variable is translated to a composite probe on a signal vector conforming to Figure 6. Two rewriting schemes are supported depending of the kind of breakpoint (Algorithms 4 and 3) as shown in Algorithm 5. The generated wrapper is used for testing the design netlist produced by Madeo+ as depicted by Figure 8.

Algorithm 5 shows the generated Verilog code for breakpoints of Algorithms 4 and 3. Declaration of signals and design instantiation are generated automatically as well.

5. Simulation Results

In order to perform verifications on the model, the designer needs tools to visualize the concurrent activities of the system under test. Our visualization tools show the system behavior as well as the interactions between components at different levels of detail. For example, at the system level, the activities are seen as tasks evolving on a time scale while at the mapped application level each clock cycle is reported.

To illustrate this, a good case study must be an application simple enough to ease explanation, that makes sense in a DSP scope and that designers do know pretty well, with no compromise on the ability to highlight the benefits our approach carries. These considerations lead us to select an

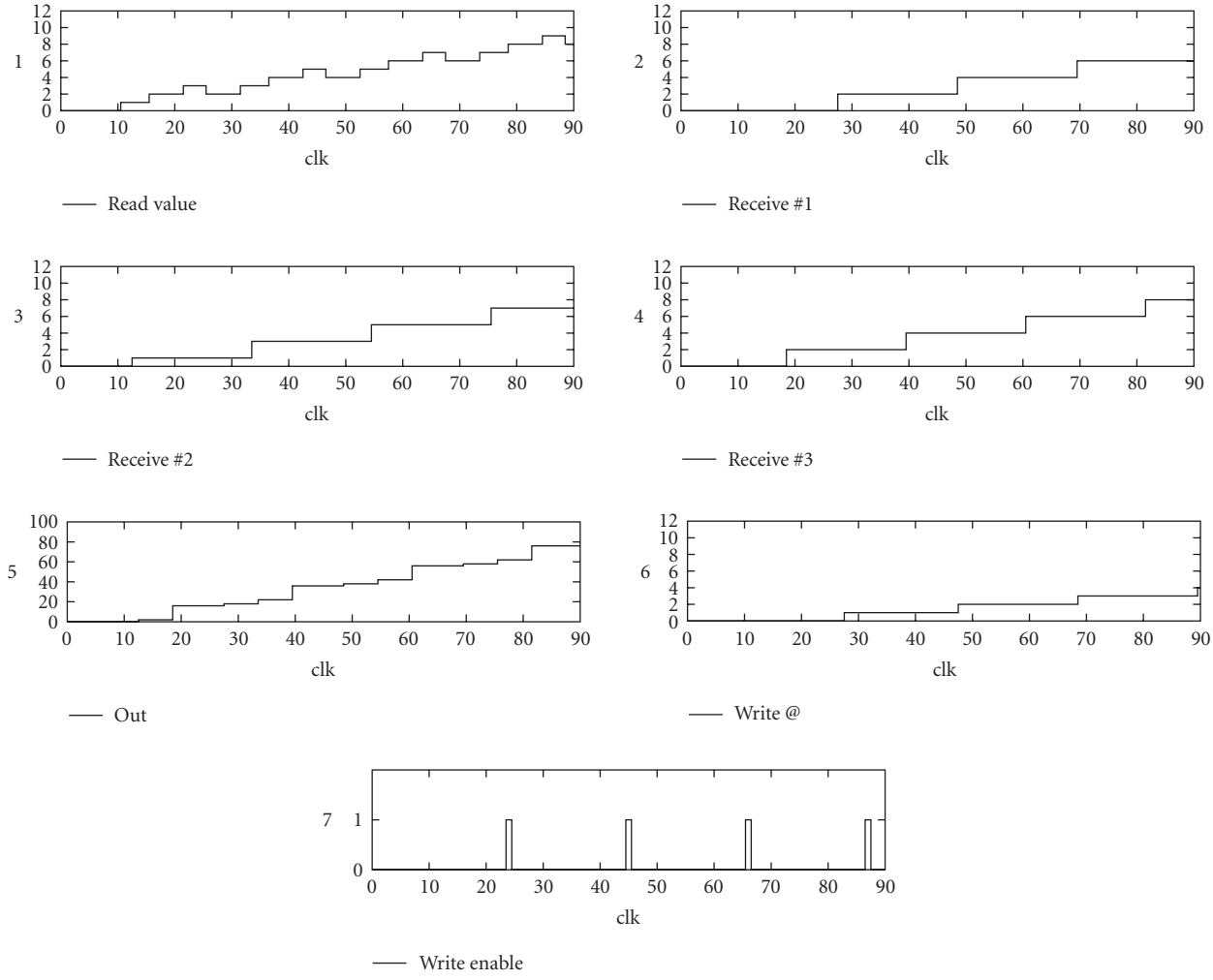

Figure 13: Internal view of a low-level simulation.

FIR filter. The global application is on purpose simplified: it is composed of two processes performing reading/writing operations on accelerator's local memories (input and output processes) for feeding an FIR filter function process and writing back results. CDFG representation of the function is given by Figure 10. The function receives three data from the input processes and sends back one result to the output process.

5.1. System-Level Simulation. The system simulator reports the behavior of each traced activities on a Gantt diagram. Figure 9 illustrates the system-level simulation of the FIR filter application running on a modeled SoC including a main memory, a bus, a DMA controller feeding accelerator's local memories, and a reconfigurable accelerator. The task names are listed on the Y-axis and the time scale on the X-axis. All the system components are modeled at a behavioral level in the SmallSystem framework.

Only the needed functionalities of components have been modeled at a high abstraction level. The processor task is to initialize the DMA and sends a start signal to the accelerator. The DMA controller performs communications in a pipelined way. The bus transfers the packets and simulates contention penalties by a probability law chosen by the designer.

The diagram of Figure 9 shows tasks related to a local execution on the reconfigurable accelerator. Traces prefixed by *BUS* show the data transfers between the main memory and the accelerator. Activities of input and output processes computing addresses for local memory accesses are reported by traces *CIn* and *COut*. Function's activities (receive, compute, send) are given by traces prefixed by *FNC*. Figure 9 focuses on the application behavior, but it is also possible to visualize system activities such as memory accesses and DMA transfers.

At this level the application is specified in Smalltalk, and the simulation is performed at software level. The refinement of the abstraction level is obtained by a translation of Smalltalk into a CDFG taken as input of the cycle-accurate simulator integrated as a component in the model (see Section 5.2).

A Gantt diagram gives a global vision of the system behavior with a fine grain tracing of the components' internal activities. Another aspect concerns the interactions

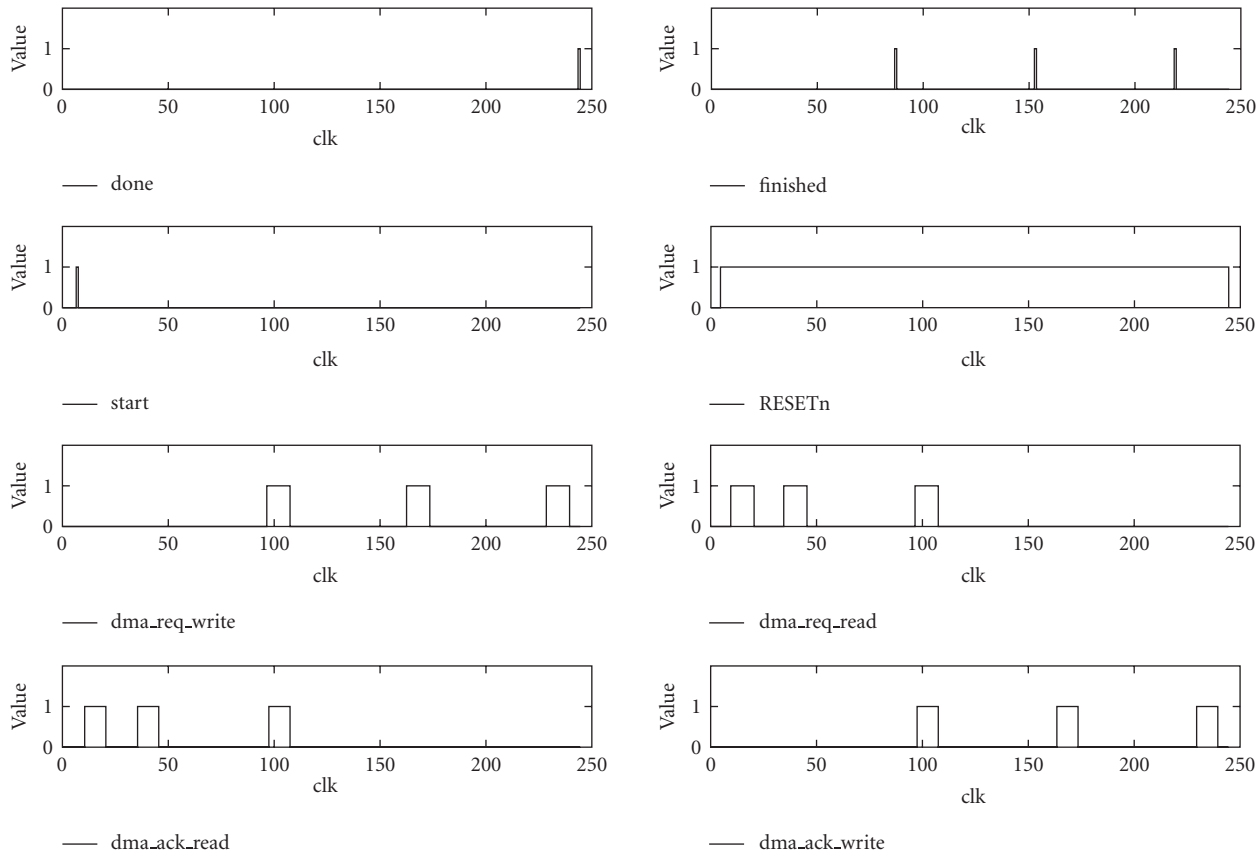

Figure 14: Low-level simulation for a computation divided in three phases.

```
initial begin
    @(posedge done); − − HALT
    $stop
end
always @(posedge dma_ack_read) begin
    #(PERIOD ∗ 10) dma_ack_read = 0;
End
```

Algorithm 5: Generated Verilog code of two conditional breakpoints.

between the components performed by communications. Communications between SoC's components are represented by an interaction diagram shown in Figure 11. A circle corresponds to a communication starting point. For example, at the time 2 the CPU sends a start signal to the DMA for initiating data transfers to the reconfigurable accelerator. Then a request is sent to the main memory (*MEM*) through the bus (*BUS*).

5.2. Cycle-Accurate Simulation of an Application. Figure 13 presents a list of traces for the low-level CDFG simulation of the previous example.

The curves represent, respectively, the value read from memory (1), the buffered values for the FIR filter computation that come from the read values at different timestamps (2) (3) (4) and correspond to the three receive operations in Figure 10, FIR filter result (5), the write address (6), and the write activation (7). Signals (1) to (6) carry numerical values while (7) is an RTL Boolean value.

Comparing Figures 9 and 13 clearly illustrates both the multilevel simulation, with as an example system activities (top three lines of Figure 9) on one side and low level and accurate values manipulation on the other side (Figure 13), and both the correlating values (line 7, Figure 9 with (7), line 9 with (2) to (4) sampling values).

5.3. Simulation with Third-Party Tools. The generated Verilog wrapper and the Madeo+ netlist are taken as input of ModelSim for final validation. Figure 12 shows application activities at the netlist level. Activities such as DMA (signals *dma*) result from interactions between the netlist and the testbench (Algorithm 4). They also appear in Figures 14 and 13. At a higher abstraction level, read/write requests are traced on Figure 9. Addresses generated by IOs processes are *ADDR0* and *ADDR2*; they also appear in Figure 13.

6. Conclusion

This paper introduces a methodology for multi level simulation of applications running on an RSoC. This work focuses on fast and early verification while preserving the ability to deeply probe the RTL model. Our debugging scheme

exploits several object-oriented facilities: agile development, test-driven design, and abstraction. Multilevel ranges from behavioral code execution to mainstream simulation engines (e.g., ModelSim) and addresses both system activities (e.g., data moves) and accelerated tasks. The successful design of significant test cases has confirmed this approach to be very valuable.

Current on-going work, related to this, addresses implementing the breakpoints as hardware primitives, with a detailed study on performances and the characterization of probe-effect in case of synthesized probes.

References

[1] A. Bernstein, M. Burton, and F. Ghenassia, "How to bridge the abstraction gap in system level modeling and design," in *Proceedings of IEEE/ACM International Conference on Computer-Aided Design (ICCAD '04)*, pp. 910–914, IEEE Computer Society, San Jose, Calif, USA, November 2004.

[2] IEEE, "Std 1076-2000: IEEE Standard VHDL Language Reference Manual," IEEE, 2000.

[3] D. E. Thomas and P. R. Moorby, *The VERILOG Hardware Description Language*, Kluwer Academic Publishers, Norwell, Mass, USA, 1996.

[4] "Open SystemC initiative," http://www.systemc.org.

[5] D. D. Gajski, J. Zhu, R. Domer, A. Gerstlauer, and S. Zhao, "The specC methodology".

[6] R. Helaihel and K. Olukotun, "Java as a specification language for hardware-software systems," in *Proceedings of IEEE/ACM International Conference on Computer-Aided Design (ICCAD '97)*, pp. 690–697, IEEE Computer Society, San Jose, Calif, USA, November 1997.

[7] A. Habibi and S. Tahar, "Design for verification of SystemC transaction level models," in *Proceedings of Design, Automation and Test in Europe (DATE '05)*, vol. 1, pp. 560–565, Munich, Germany, March 2005.

[8] T. Kuhn, T. Oppold, M. Winterholer, W. Rosenstiel, M. Edwards, and Y. Kashai, "A framework for object oriented hardware specification, verification, and synthesis," in *Proceedings of the 38th Design Automation Conference (DAC '01)*, pp. 413–418, Las Vegas, Nev, USA, June 2001.

[9] M. C. W. Geilen, J. P. M. Voeten, P. H. A. van der Putten, L. J. van Bokhoven, and M. P. J. Stevens, "Object-oriented modelling and specification using SHE," *Computer Languages*, vol. 27, no. 1–3, pp. 19–38, 2001.

[10] S. Vernalde, P. Schaumont, and I. Bolsens, "An object oriented programming approach for hardware design," in *Proceedings of IEEE Computer Society Workshop on VLSI (WVLSI '99)*, Orlando, Fla, USA, April 1999.

[11] S. Swamy, A. Molin, and B. Covnot, "OO-VHDL: object-oriented extensions to VHDL," *Computer*, vol. 28, no. 10, pp. 18–26, 1995.

[12] "Extrem programming methodology," http://www.extreme-programming.org.

[13] L. Lagadec, J. Boukhobza, and A. Plantec, "Chaîne de programmation pour architecture hétérogène reconfigurable," in *Le Prochain Symposium en Architecture de Machines (SympA '08)*, 2008.

[14] A. Goldberg and D. Robson, *Smalltalk-80: The Language and Its Implementation*, Addison-Wesley, Boston, Mass, USA, 1983.

[15] "Visualworks smalltalk," http://www.cincom.com.

[16] M. Feathers, *Working Effectively with Legacy Code*, Prentice Hall PTR, Upper Saddle River, NJ, USA, 2004.

[17] S. R. Alpert, K. Brown, and B. Woolf, *The Design Patterns SmallTalk Companion*, Addison-Wesley, Boston, Mass, USA, 1998.

[18] C. A. R. Hoare, "Communicating Sequential Processes," 1985.

[19] Part 11: edition 2, "EXPRESS Language Reference Manual," ISO 10303-11, 2004.

[20] "Platypus Technical Summary and download," 2007, http://cassoulet.univ-brest.fr/mme.

[21] S.-T. M. Limited, OCCAM 2.1 reference manual, 1995.

[22] "Modelsim," http://www.model.com.