A TEST INFRASTRUCTURE FOR COMPILERS TARGETING FPGAS

Rui M. M. Rodrigues[1]
and
João M. P. Cardoso[1,2]

[1]*Faculty of Sciences and Technology,*
University of Algarve,
Campus de Gambelas, 8000 – 117 Faro, Portugal
[2]*INESC-ID, Lisbon,*
Email: jmpc@acm.org

ABSTRACT

This paper presents an infrastructure to verify the functionality of the specific architectures generated by a high-level compiler, targeting dynamically reconfigurable hardware. Java, XML, and XSL technologies are used to support the infrastructure. As simulation engine we use Hades, an event driven Java based simulator.
It results in a suitable scheme to test the designs generated by the compiler each time a new optimization technique is included or changes in the compiler are performed. We believe this infrastructure will be very important to verify, by functional simulation, further research techniques, as far as compilation to FPGA-based reconfigurable computing is concerned.

KEYWORDS

Reconfigurable computing, Hardware Compilers, RTL-Simulation, Datapath, Control Units, FPGA

1. INTRODUCTION

Mapping software languages onto reconfigurable hardware structures (*e.g.*, FPGAs) is an important research subject (see, for instance, [1][2]). It promises both to shrink the long design cycle needed to develop systems using reconfigurable hardware components and to take advantage of the large number of gates available in the new FPGA devices.

At the front-end level, researchers have used compiler frameworks such as SUIF [3]. However, in order to test new techniques they usually use a HDL simulator or go directly to the implementation on the FPGA. Both test flows are not appropriate to conduct an efficient high-level test, which requires among other issues automation mechanisms. Although going to the implementation of the architecture on the FPGA permits a fast execution of the reconfigurable hardware structures, it requires long design cycles due to the need to perform logic synthesis and place and route. Furthermore, testing by implementation imposes other difficulties such as the access to values on certain connections, the lack of assertions, the inclusion of probes and stop mechanisms, the automation needed to test the results for all the set of test cases used during the test, etc.

When simulating algorithms, requiring large amounts of data (e.g., image and video processing), the large number of clock cycles needed to simulate requires a fast simulation engine. Previous work has shown that RTL simulation based on software languages can be faster than commercial HDL simulators [4][5].

Implementations with several configurations need an appropriate scheme to simulate the flow of configurations and the communication between them. Since HDLs have not been developed for programming dynamically reconfigurable hardware devices, they do not directly support those features. Trying to solve these issues, tools for simulation of partial runtime reconfigurable hardware have already been presented (e.g., [6]).

Previous work on research and developing a compiler for FPGA-based reconfigurable platforms [7] has revealed difficulties related to verification of designs output by the compiler, each time modifications are carried on. Those modifications may include code restructuring or addition of new optimization techniques. Checking the overall test suite required long time efforts.

Furthermore, the need to couple the hardware reconfigurable component to a microprocessor model might be required and thus a hardware/software simulation may be important. Note also that using the same language may be important to mix the software part of the algorithm with the reconfigurable hardware components without specialized co-simulation environments [5].

To alleviate all these problems we developed an infrastructure we believe is an efficient scheme to verify, by functional simulation, research efforts on compilation techniques. The infrastructure is currently being used in the Galadriel and Nenya compiler [7]. To accomplish our goals, the infrastructure takes advantage of the XML (eXtensible Markup Language) [8], XSL (eXtensible Stylesheet Language) [9], and Java technologies. XML, a meta-language, has been used to describe the structure of an application and architecture models [10], finite state machine behaviors [11], instruction set architectures [12], structure of reconfigurable arrays and dataflow implementations [13], etc. With a built-in XML transformation engine (XSLT), it furnishes a suitable technology for incremental development of languages and associated translation mechanisms. Languages based on XML are best suited as intermediate representation schemes since their human readability is very difficult, at least when representing complex structures. In our case XML-based languages are used to represent structures and behaviors generated by a compiler.

This paper is organized as follows. Section 2 explains in some detail the infrastructure and the XML-based languages being used. Section 3 presents some preliminary tests and results. Finally, last section presents concluding remarks.

2. ENVIRONMENT

The infrastructure (see Figure 1) uses as simulation engine Hades [14][15]. Hades is an event-based simulator developed in Java. Since it receives behavioral Java descriptions, it permits to take profit of all the Java features. Using this capability, we have developed a library of operators written in Java to be used with Hades. Such library permits to simulate the architecture under functional test. The library is composed by several elements reproducing the behavior of each operator existent in the compiler library, used to describe the datapath (adders, registers, RAMs, etc.). Additional operators have been developed to perform actions such as signal data tracing and simulation control.

The Galadriel and Nenya compiler [7] has been modified to output the specification of the datapaths, control units, and the reconfiguration transition graph (RTG), using XML dialects. Those XML files are then translated to the required language by XSLT engines. This is useful because it permits a user to define its own XSL translation rules to output each representation in an adequate language (Verilog, VHDL, etc.). Note that the RTG is used when the compiler implements the input algorithm using multiple configurations (temporal partitions) [7].

As far as the test infrastructure is concerned, the XML representation of the datapaths are translated to the Hades input format, the XML representations of the behavioral descriptions of the FSMs (Finite State Machines) to Java code representing their behavior, and the RTG to Java code that controls the execution of the simulation through the set of designs (each one related to a temporal partition) under simulation.

With the intention of simple coarse debugging the infrastructure also translates the XML files to descriptions to be seen with Graphviz [16], a graph visualization tool.

The test process automation is acquired by using Ant [17] build files that execute the set of tasks shown in Figure 1. The test infrastructure also reports automatically the total number of clock cycles to execute a certain design, among other features.

Memory contents and I/O data are stored in files. Those files are used when executing the Java input algorithm using a specific Java class with methods to read/write from/to those files. After simulation, a simple comparison of data content is performed to verify results.

As it has been previously referred, the datapath, the control unit, and the RTG are described using XML dialects. Next, we explain the XML dialects adopted to specify each one of those elements.

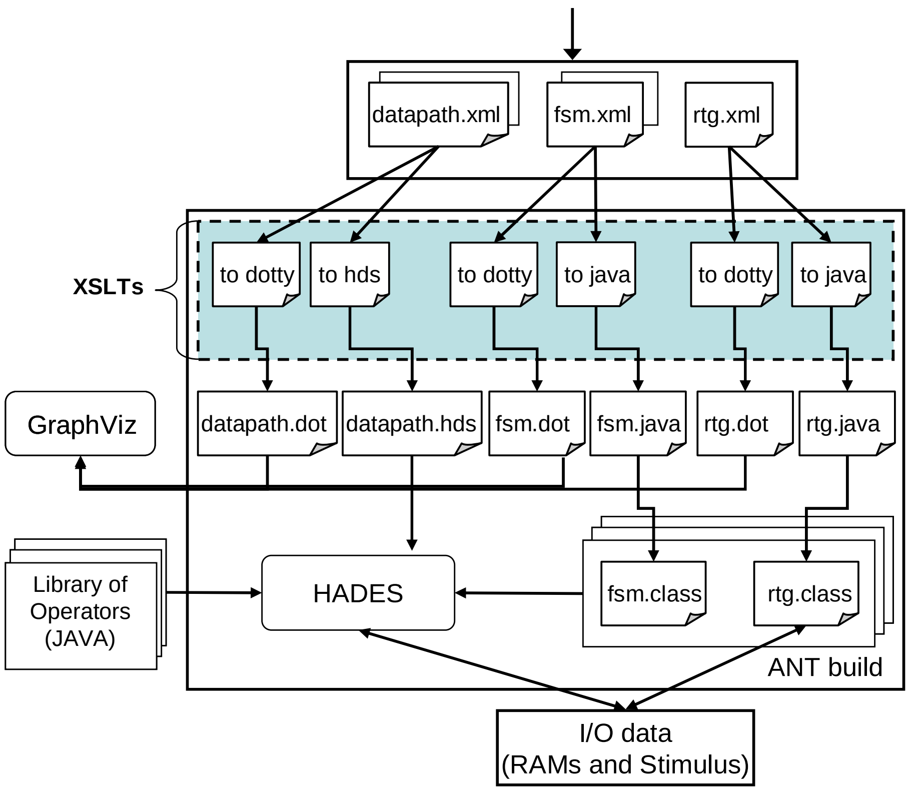

Figure 1. Diagram of the test infrastructure. Inputs are XML specifications. XSLT transformation engines are used to translate those XML files to Java and Hades formats. I/O data are stored in files. The Ant framework is used to automate the flow infrastructure in order to test one or the set of benchmarks used to test the compiler.

2.1. Finite State Machine Specification

Each temporal partition has a control unit, described as an FSM. Part of an example of the XML dialect that describes a control unit is shown in Figure 2. The statement begins with the element root FSM and its name. Next the INTERFACE is described, with all the FSM PORTS used to communicate with the datapath. Each element PORT can be an input port (*type=”in”*) or an output port (*type=”out”*). The *feature* attribute specifies special signals such as *clock* and *reset*. The next FSM first order child element is the element BEHAVIOR and is were the control unit behavior is described. The description begins with possible default constant assignments to signals, element SET (those assignments are executed for all states, however if local state specification overrides them, the last assignments prevails). Next the declaration of each STATE behavior is specified. The initial state is specified with the attribute *type=”reset”* and inside each STATE is possible to perform a set of different actions, being the simplest one the assignment of values to signals using the SET element. When the transition depends on specific conditions the IF/ELSE control flow is allowed. In this situation, the element TEST is used to describe a transition condition. If more than one condition is needed OR and AND logical operations can be used. Finally, the NEXT element indicates the next FSM state.

Each FSM specified in XML is then translated to a Java class representing the same behavior. This Java class is used by Hades to simulate the design. For prototyping the design, the XML is translated to RTL behavioral VHDL, ready for logic synthesis. Each translation is performed by a specific XSL transformer as can be seen by the flow diagram depicted in Figure 1.

```
<?xml version="1.0" encoding="UTF-8"?>
<FSM name="GCDstg0">

  <INTERFACE>
    <PORT type="out" signal="ld_reg_op_1"/>
    <PORT type="out" signal="done" feature="done"/>
    <PORT type="in" signal="flag_10"/>
    <PORT type="in" signal="start" feature="start"/>
    <PORT type="in" signal="clock" feature="clock"/>
    <PORT type="in" signal="reset" feature="reset"/>
  </INTERFACE>

  <BEHAVIOR>
    <SET signal="done" value="0"/>
    …
    <STATE name="State_reset" type="reset">
      <NEXT name="State1"/>
    </STATE>

    <STATE name="State1">
      <IF>
        <AND>
          <TEST signal="start" operation="=" value="1"/>
          <TEST signal="reset" operation="=" value="0"/>
        </AND>
        <NEXT name="State2"/>
      </IF>
      <ELSE>
        <NEXT name="State1"/>
      </ELSE>
    </STATE>
    …
    <STATE name="State4">
      <SET signal="done" value="1"/>
      <NEXT name="State1"/>
    </STATE>
  </BEHAVIOR>
</FSM>
```

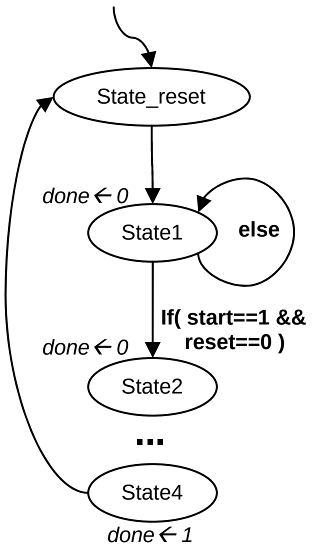

Figure 2. Part of an FSM XML specification example. The picture on the right depicts part of the FSM state transition graph.

2.2. Datapath Specification

The datapath for each temporal partition consists of a structure of components from the library and the interconnections between those components. Part of an example of the XML dialect that describes a datapath is shown in Figure 3. It begins with the element root DESIGN and such as in the FSM XML specification it is followed by the child element INTERFACE, where the *clock* signal is declared, the existent control signals (*start, reset* and *done*), and all other input/output datapath signals. Next are described all the components in a datapath (element COMPONENT). The attribute *unit* specifies the correspondent operator in the library. If component input and output ports (other than *clock* or *control signal* ports) have the same bitwidth then, a single attribute *bitwidth* is used. If the ports have different bitwidths or at least one has as input a constant value, the child element PORT is declared for each one of the ports and individual bitwidth values are declared. To assign a constant value to an input port, the attribute *value* in element PORT is used.

The connections between ports of two components are specified using SIGNAL elements. Each element indicates the SOURCE component port, and one or more SINK component ports (destinations). Each SIGNAL indicates the *bitwidth* that correspond to the bitwidth of the SOURCE and SINK ports.

All the *out* type signals described in the INTERFACE element can be traced out to a file. However, if we are interested to check the data on any signal, an element named PROBE, a child of element SINK, can be included.

```
<?xml version="1.0" encoding="UTF-8"?>
<DESIGN name="GCD">

  <INTERFACE>
    <PORT type="in" signal="reset" name="reset" feature="reset"/>
    <PORT type="in" signal="start" name="start" feature="start"/>
    <PORT type="in" signal="clock" name="clock" feature="clock"/>
    <PORT type="in" name="in_7" bitwidth="32" signal="in_7"/>
    <PORT type="out" signal="done" name="done" feature="done"/>
  </INTERFACE>

  <COMPONENT unit="stg" name="GCDstg0">
    <PORT type="in" signal="reset" name="reset" feature="reset"/>
    <PORT type="in" signal="start" name="start" feature="start"/>
    <PORT type="in" signal="clock" name="clock" feature="clock"/>
    <PORT type="out" signal="done" name="done" feature="done"/>
  </COMPONENT>

  <COMPONENT unit="gt_op_s" name="gt_op_s" bitwidth="32"/>

  <COMPONENT unit="reg_op" name="reg_op_1" bitwidth="32">
    <PORT type="in" signal="clock" name="Ck" feature="clock"/>
   </COMPONENT>
   …
  <SIGNAL name="sinal_flag_10" bitwidth="1">
    <SOURCE name="gt_op_s" port="O0"/>
    <SINK name="GCDstg0" port="flag_10">
      <PROBE/>
    </SINK>
  </SIGNAL>

  <SIGNAL name="n5" bitwidth="32">
    <SOURCE name="reg_op_1" port="O0"/>
    <SINK name="mux_op_4" port="I1"/>
    <SINK name="if_ge_op_s" port="I0"/>
    <SINK name="gt_op_s" port="I0"/>
    <SINK name="mux_op_2" port="I0"/>
   </SIGNAL>
   …
```

```
   …
   <SIGNAL name="sinal_st1_9" bitwidth="1">
     <SOURCE name="GCDstg0" port="st1_9"/>
     <SINK name="reg_op_1" port="Ld"/>
   </SIGNAL>
</DESIGN>
```

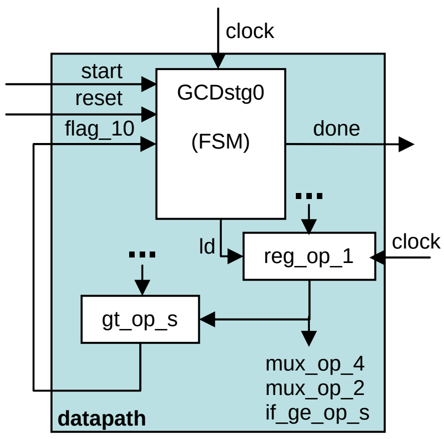

Figure 3. Part of the datapath XML specification for an example. The picture on the right depicts part of the datapath structure.

2.3. Reconfiguration Transition Graph (RTG) Specification

The RTG represents the behavior of the configuration flow when one application is mapped to more than one temporal partition. When advanced reconfiguration mechanisms are employed, the RTG may assemble cyclic flow in order to implement loops in the original program that have been partitioned (see, for instance, [18] and [19]). The RTG behavior is usually implemented using a microprocessor, a specific FSM, or by a configuration manager existent in advanced reconfiguration architectures.

Figure 4 shows part of an example of RFG XML specification. Its description begins with the element root RTG, and is followed by one child element SET that specifies the starting CONFIGURATION to be simulated. Each CONFIGURATION element is composed by the attributes *name* and *design* (the HADES netlist file to be simulated). The behavior is similar to FSM STATE elements. NEXT indicates the next configuration to be executed. When a transition between configurations is conditional, elements IF/ELSE, OR, and AND can be used. The element TEST has the attribute *file* that indicates the RAM output data file where a specific *address* should be read to perform the TEST condition. This means that our target architectures use RAMs to communicate data between configurations.

The stop condition is established when a CONFIGURATION stops execution and no NEXT element has been specified.

As has been previously referred, for simulating a flow of configurations a XSLT transformer has been developed that translates the XML RFG into Java code responsible to execute the simulator for each configuration and to orchestrate the flow of configurations. Such behavior represents the high-level behavior of a typical configuration manager. Figure 5 presents part of the java code generated from the XML specification in Figure 4. For each configuration, an external application that simulates the correspondent architecture is called. That application is a simple class that receives as parameter the netlist file for Hades (*.hds*), and executes a Hades batch simulation that runs until a stop event occurs.

```
<?xml version="1.0" encoding="UTF-8"?>

<RTG name="Example">
   <SET start="config0"/>

   <CONFIGURATION name="config0" design="name1.hds">
      <NEXT name="config1"/>
   </CONFIGURATION>

   <CONFIGURATION name="config1" design="name2.hds">
      <IF>
         <TEST file="ram.out" address="23" operation="==" value="1"/>
         <NEXT name="config0"/>
      </IF>
      <ELSE>
         <NEXT name="config2"/>
      </ELSE>
   </CONFIGURATION>

   <CONFIGURATION name="config2" design="name3.hds"/>
</RTG>
```

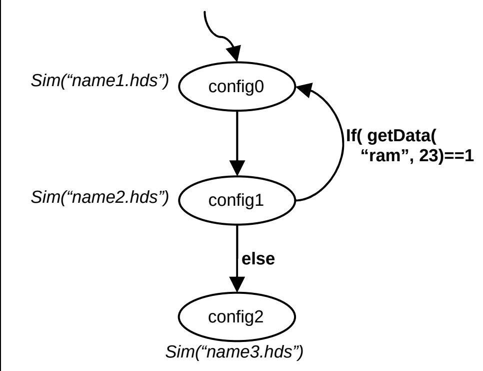

Figure 4. Example of an RFG XML specification. The picture on the right depicts part of the RTG.

```
...
public class Example {
   …
   int state = state_config_0;
   boolean proc;
   String[] config0 = { "cmd.exe", "/c", "java –Xms256M RunHds name1.hds" };
   String[] config1 = { "cmd.exe", "/c", "java –Xms256M RunHds name2.hds" };
   String[] config2 = { "cmd.exe", "/c", "java –Xms256M RunHds name3.hds" };
   try {
      while (run) {
         switch (state) {

            case state_config0:
               proc = Runtime.getRuntime().exec(config0);
               exitVal = proc.waitFor();
               state = state_config1;
               break;

            case state_config1:
               proc = Runtime.getRuntime().exec(config1);
               exitVal = proc.waitFor();
               if ((IO.getData("ram..out", 23)) == 1) {
                  state = state_config0;
               } else {
                   state = state_config2;
               }
               break;

             case state_config2:
                proc = Runtime.getRuntime().exec(config2);
                exitVal = proc.waitFor();
                run = false;
                break;
   …
}
```

Figure 5. Part of the Java code simulating the behavior of the RTG shown in Figure 3.

As can be seen in the Java code, the communication of data between configurations is performed using RAMs. The simulation stops when variable *run* assumes value *false* indicating the final configuration to be executed.

3. EXPERIMENTAL RESULTS

Table I shows results with a fast DCT (Discrete Cosine Transform) algorithm, FDCT, and with a Hamming decoder. The simulation time presented corresponds to the execution of the simulation in a Pentium 4, 2.8 GHz, with 512 MB of RAM, under Windows XP. The table shows the complexity of the XML and Java files, in number of lines of code, used as input specifications to the environment. The column "Operators" represents the number of functional units used for each datapath. Last two columns show the number of lines of VHDL code generated by translation of the input XML specifications using XSL engines.

Table I. Results using the test infrastructure.

Example	loJava	loXML FSM	loXML datapath	loJava FSM	#Operators	Simulation time (s)	loVHDL FSM	loVHDL datapath
FDCT1	138	512	1,708	1,175	169	6.9	496	1,752
FDCT2	138	258 256	860 891	667 606	90 90	2.9 2.9	299 253	1,106 1,117
Hamming	45	38	322	134	37	1.5	48	683

The FDCT performs 8 × 8 DCT blocks over an input image. The results present two implementations of the algorithm. The first one, FDCT1, with only one configuration (i.e., the algorithm is implemented in a single datapath and control unit) and the second one, FDCT2, with two configurations (temporal partitions). In the later case, the implementations use two separate designs that respect the functionality of the original application by executing the two configurations in sequence. Both implementations use three SRAMs to store input, output, and intermediate images.

The simulation time of the FDCTs is related to the computation on an input image of 4,096 pixels (64 DCT blocks). With images of 65,536 and 345,600 pixels, the simulation of the FDCT requires 1 and 6.5 minutes, respectively.

4. CONCLUSIONS

We have presented an infrastructure to verify the functionality of the specific architectures output by a high-level compiler targeting dynamically reconfigurable hardware. The environment is able to simulate the flow of configurations that may exist when more than one temporal partition is used. From the presented results we may conclude that the test environment is fast enough to our purposes. We have obtained an efficient and high-level automation mechanism that permits to verify the compilation results over a complete test suite in feasible time.

We expect that this infrastructure helps us to further research new optimization techniques for mapping computational structures to FPGA-based reconfigurable computing platforms.

ACKNOWLEDGEMENTS

This work is partially supported by the Portuguese Foundation for Science and Technology (FCT) - FEDER and POSI programs - under the CHIADO project (POSI/CHS/48018/2002). The authors also acknowledge Ricardo Ferreira for showing them the Hades simulator and for some fruitful discussions about it.

REFERENCES

[1] Byoungro So, Mary Hall and Pedro Diniz, “A Compiler Approach to Fast Design Space Exploration in FPGA-based Systems,” In *Proc. of the ACM Conference on Programming Language Design and Implementation (PLDI'02)*, ACM Press, New York, June, 2002.

[2] João M. P. Cardoso, Pedro Diniz, and Markus Weinhardt, “Compilation for Reconfigurable Computing Platforms: Comments on Techniques and Current Status,” *submitted to the ACM Computing Surveys (September 2002)*. INESC-ID Technical Report 40/2003, Lisbon, Portugal, October 2003.

[3] S. P. Amarasinghe, J. M. Anderson, M. S. Lam and C. W. Tseng, “The SUIF Compiler for Scalable Parallel Machines,” In *Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computing*, February, 1995. URL: http://suif.stanford.edu/

[4] T. Kuhn, W. Rosenstiel, and U. Kebschull, “Object Oriented Hardware Modeling and Simulation Based on Java,” in *International Workshop on IP Based Synthesis and System Design*, Grenoble, France, 1998.

[5] Luc Séméria, et al., “RTL C-Based Methodology for Designing and Verifying a Multi-Threaded Processor,” in *Proc. of Design Automation Conference (DAC’02)*, June 2002, New Orleans, USA.

[6] I. Robertson, and J. Irvine, “A Design Flow for Partially Reconfigurable Hardware,” in *ACM Transactions on Embedded Computing Systems*, Vol. 3, No. 2, May 2004, pp. 257–283.

[7] João M. P. Cardoso, and Horácio C. Neto, “Compilation for FPGA-Based Reconfigurable Hardware,” in *IEEE Design & Test of Computers Magazine*, March/April, 2003, vol. 20, no. 2, pp. 65-75.

[8] W3C: *Extensible markup language (xml)*, http://www.w3.org/XML/ (1996-2003).

[9] W3C: *The extensible stylesheet language family (XSL)*, http://www.w3.org/Style/XSL/.

[10] J. E. Coffland and A. D. Pimentel, ``A Software Framework for Efficient System-level Performance Evaluation of Embedded Systems", in *Proc. of the 18th ACM Symposium on Applied Computing (SAC’03)*, Melbourne, Florida, USA, March 2003, pp. 666-671.

[11] Eric Keller, Gordon J. Brebner, Philip James-Roxby, "Software Decelerators," in *Proc. of the 13th International Conference on Field Programmable Logic and Applications (FPL’03)*, Lisbon, Portugal, Sept. 2003. Peter Cheung, George Constantinides, José T. Sousa (Editors), LNCS 2778, Springer Verlag, pp. 385-395.

[12] Shay P. Seng, Krishna V. Palem, Rodric M. Rabbah, Weng-Fai Wong, Wayne Luk and P.Y.K. Cheung, “PD-XML: Extensible Markup Language For Processor Description,” In *Proceedings of the IEEE International Conference on Field-Programmable Technology* (ICFPT), December 2002, pp. 437- 440.

[13] Ricardo Ferreira, João M. P. Cardoso, and Horácio C. Neto, “An Environment for Exploring Data-Driven Architectures,” in *14th International Conference on Field Programmable Logic and Applications (FPL’04)*, Antwerp, Belgium, August 30 - September 1, 2004, LNCS 3203, Springer-Verlag, Jürgen Becker, Marco Platzner, Serge Vernalde (eds.), August 2004, pp. 1022 - 1026.

[14] Norman Hendrich, “A Java-based Framework for Simulation and Teaching,” in *Proc. of the 3rd European Workshop on Microelectronics Education (EWME’00)*, Aix en Provence, France, 18-19, May 2000, Kluwer Academic Publishers, pp. 285-288.

[15] *Hades simulation framework homepage*: http://tech-www.informatik.uni-hamburg.de/applets/hades/html/hades.html.

[16] *Graphviz - open source graph drawing software*: http://www.research.att.com/sw/tools/graphviz/.

[17] The apache ant project: http://ant.apache.org/.

[18] João M. P. Cardoso, “Loop Dissevering: A Technique for Temporally Partitioning Loops in Dynamically Reconfigurable Computing Platforms,” in *10th Reconfigurable Architectures Workshop (RAW 2003), 17th Annual International Parallel & Distributed Processing Symposium (IPDPS 2003)*, Nice, France, April 22, 2003.

[19] João M. P. Cardoso, and Markus Weinhardt, “From C Programs to the Configure-Execute Model,” in *Proc. of the Design, Automation and Test in Europe Conference (DATE’03)*, Munich, Germany, March 3-7, 2003, IEEE Computer Society Press, pp. 576-581.