Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2009, Article ID 205149, 9 pages
doi:10.1155/2009/205149

Research Article

vMAGIC—Automatic Code Generation for VHDL

Christopher Pohl, Carlos Paiz, and Mario Porrmann

Heinz Nixdorf Institute, University of Paderborn, Fürstenallee 11, D - 33102 Paderborn, Germany

Correspondence should be addressed to Christopher Pohl, pohl@hni.upb.de

Received 28 November 2008; Accepted 26 March 2009

Recommended by Michael Huebner

Automatic code generation is a standard method in software engineering, improving the code reliability as well as reducing the overall development time. In hardware engineering, automatic code generation is utilized within a number of development tools, the integrated code generation functionality, however, is not exposed to developers wishing to implement their own generators. In this paper, VHDL Manipulation and Generation Interface (vMAGIC), a Java library to read, manipulate, and write VHDL code is presented. The basic functionality as well as the designflow is described, stressing the advantages when designing with vMAGIC. Two real-world examples demonstrate the power of code generation in hardware engineering.

1. Introduction

The notion *automatic code generation* (ACG) envelopes a number of different techniques aimed at simplifying the task of writing a code. Apart from specific implementation details these techniques differ in the level of abstraction exposed to the developer: a very low level of abstraction is given by template-based techniques such as code completion or code insertion. These allow for the generation of code structures with a low complexity (e.g., getters/setters), which are inserted into the code by the user on an explicit (calling an editor function) or implicit (the editor recognizes the beginning of a construct and completes it) basis. This is a very general approach that can be applied in every programming language and in any kind of desired application. Code transformation represents a higher level of abstraction, where a piece of code is translated from a source language into a target language. This is useful if domain specific language (DSL) exists to describe features of a very specific kind of application, which will be implemented in another, more general language. This approach can obviously save a lot of time, as the definition of an application in the very specific and abstract DSL typically takes much less time than implementing it in the specific target language.

Code generators such as the aforementioned exist for various types of applications in computer science, for example, parser generators, database generators, or unified modeling language (UML) tools, rapidly generating production code and saving development costs. Apart from saving time by generating code which would otherwise have to be implemented manually, (correct) generators deliver correct code; additionally, all developmental iterations (with alterations to the specification) can be handled in the source language, again saving time. While these principles and methods are largely applied in the area of software development, hardware developers are supported by very few specific tools like IP-Core Generators or C-to-hardware compilers, covering only a very small area of what could be done with ACGs. Examples for C-based or graphical tools can be found in [1–4]. Each of these tools utilizes code generators for some hardware description language in their specific area, however, this functionality is not exposed to a developer wishing to implement own code generator.

VHDL Manipulation and Generation Interface (vMAGIC) has been developed to fill in this gap by providing a basis for all kinds of VHDL code generators by implementing three important basic tasks:

(i) reading of existing code,

(ii) manipulation of existing code, generation of new code,

(iii) writing of manipulated and/or generated code.

vMAGIC is not a code generator by itself, but a homogenous framework for implementing code generators (and other applications, cf. Section 2). Using vMAGIC accelerates design processes whenever uniform tasks can be automated and reused many times. vMAGIC is free software under LGPL 3 and can be obtained via http://vmagic.sourceforge.net/.

The general concepts and the user interface to the vMAGIC API are discussed in Section 2, as are a very small example and a number of possible applications beyond code generators. In Sections 3 and 4, two examples (HiLDE and HiLDEGART) are provided to demonstrate the power of vMAGIC. This paper concludes with a summary the benefits of the presented approach, and the work in progress is shown.

2. Automatic Code Generation with vMAGIC

The description of the vMAGIC API starts with the implementation of the main functionality of parsing, modifying, and writing VHDL code. The next section is devoted to the API comprising of a set of so-called metaclasses. The description of vMAGIC concludes with an example and an outlook on further use cases for vMAGIC.

2.1. Functionality. The vMAGIC API is a Java library compatible with runtime environments 1.5 and later. Therefore, it is platform independent and usable in command line tools, graphical user interfaces, and scripts. The full functionality is given in the API documentation [5], here the implementation of the basic functionality is described.

(i) Parser. vMAGIC implements a VHDL'93 compliant parser to transform VHDL code into a more convenient internal representation called Abstract Syntax Tree (AST). An AST in general contains all information from the code, while redundancy (parenthesis, semicolons, and so on are implicitly included in the tree structure) is removed; this AST however is shaped in a way optimally suited for the manipulations described next.

The vMAGIC parser was generated using ANTLR 3.1 [6], a powerful parser generator; parsers generated with ANTLR support certain error recovery strategies. This implies that the vMAGIC parser can correct certain unambiguous syntactical errors while parsing VHDL source code, such as additional semicolons in a port or component.

(ii) Modification and Generation of Code. The AST by itself is a tree structure, which is not well suited for human interaction. To hide this structure behind a simple API, a set of so-called metaclasses was defined. Metaclasses combine the functionality to generate or modify specific VHDL constructs and the knowledge how to interact with the AST, such that the developer is using a homogenous API with intuitive functions. For example, `Signal.getIdentifier()` returns the identifier of a signal, `new Process()`creates a new VHDL process.

Objects of metaclasses are created either by the user, defining a VHDL design from scratch, or using a VHDL template. The template is parsed into an AST, which is then parsed by a tree parser generating the metaobjects and discarding the original tree. This approach is, again assuming that the tree grammar is correct, another means of ensuring that the generated code is correct regardless of coding style or context. The metaobjects implicitly define a descendible tree structure, beginning with a `VhdlFile` object with members for `Entity` and `Architecture` objects and so on. User programs work on this metatree rather than on the AST, allowing for intuitive software development for hardware generation or analysis purposes.

There are two different levels of abstraction represented by metaclasses: the so-called low-level classes represent basic VHDL constructs such as signal declarations or processes; the high-level metaclasses combine several low-level classes such as to create complex functionality like registers or state machines. The use of high-level classes implies a higher level of abstraction and therefore an improved coding speed.

(iii) VHDL Writer. To generate VHDL code from an AST, a VHDL Writer based on ANTLR's String Template system was developed. Again, a tree parser is used to analyze the tree and templates are used to generate VHDL code constructs. These templates are defined in a single text file in a very simple format, such that the developers preference in coding style (e.g., the use of lower case or upper case letters for keywords, or using optional identifiers at the end of a process or entity) are implementable by changing this text file.

The vMAGIC designflow, as depicted in Figure 1, follows the three steps as described above.

2.2. vMAGIC API. The complete API functionality of vMAGIC is contained in the metaclasses, accessible via member functions. The most important functions not related to VHDL are `public String toVhdlString()` implemented by all metaclasses and the `public static VhdlFile parse()` function in the VhdlFile class. The `toVhdlString()` function generates VHDL code from every possible VHDL element, such that either the complete design or parts of the design can be viewed as VHDL code (cf. II-C, where VHDL is generated for an architecture). The `parse()`function is an overloaded function, allowing for the creation of VhdlFile objects from various sources. The usage of both functions is demonstrated in Example 1. All other members are VHDL related and can be found in the vMAGIC API-Documentation on the vMAGIC website.

Another important feature of the API is the possibility to process so-called vMAGIC-tags, beginning with "$-*$" in the source code. This is a means to pass additional information from the source code to a vMAGIC application, for example, categorizing files or processes, or giving instructions on what to do with certain parts of the design.

Extracting information from a VHDL-file in vMAGIC is based on high-level functions accessing the AST: generating new code means creating and joining metaobjects. Using the vMAGIC API ensures syntactically correct code. In the next section, the use of metaclasses is demonstrated by means of an example.

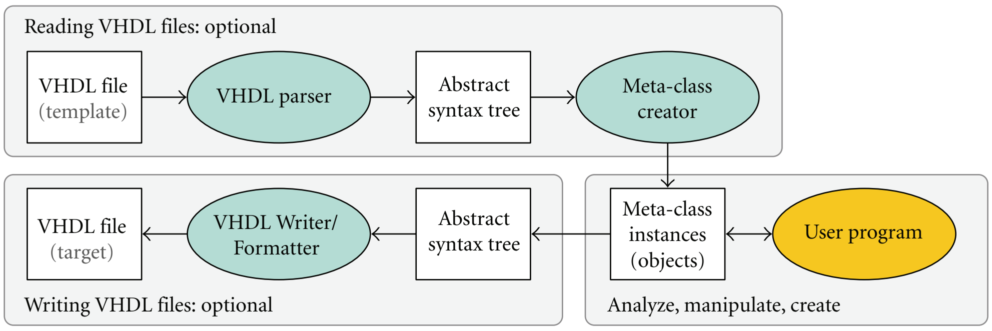


FIGURE 1: vMAGIC Designflow, reading and writing VHDL is optional. As such a vMAGIC application can be a pure VHDL generator or analyzer.

2.3. Example. The code listed in Algorithm 1 constructs a very simple wrapper file for an entity in the VHDL file *test.vhd*.

In lines 1 et seq. the input file is parsed, the entity is extracted and an output file with the same entity is generated. Lines 5–10 create an architecture *beh* with the component declaration of the entity and the instantiation of and add it to the output file. The `for` loop in lines 11–25 generates registers for all signals in the entity of : line 12 et seq. generate and declare an internal signal of the same type as the port signal, which is then connected to a new register based on the mode of the port signal. If the port signal is an input to (see line 15), the register connects the appropriate signal of the wrapper entity with the intermediate signal (line 16), which is then connected to the instance of (line 17). The registers for output ports are generated in line 18 et seq. After the loop has finished by adding the register to the architecture (line 22), the reset and clock signals are generated and added to the wrapper entity, and the code is printed out.

The output for an entity called , containing one 8 Bit input and one 8 Bit output signal, is given in Algorithm 2. For the tiny entity , creating this very simple code generator takes about twice as much time as creating the wrapper file manually. However, the wrapper generator can be reused on every possible entity, thus saving a lot of time even for this very simple case.

2.4. Advantages and Application Areas. Using a code generator always implies spending additional time on the code generator rather than programming an application manually. However, if the code generator is versatile, and this degree of versatility cannot be reached using methods native to the target language (e.g., Generics in VHDL), then it is very likely that the time spent on the generator is less than the time saved by using it. In VHDL, that point is easily reached due to limitations of VHDL as a computer language: everyday objects such as multiplexers with a generic number of inputs or busses with a generic number of devices cannot be described efficiently in standard synthesizable VHDL. Apart from these limitations, the implementation of DSLs can greatly improve coding speed as shown in the next section.

Apart from code generation, vMAGIC provides important mechanisms to extract information such as signal names, design hierarchies, or generic values from VHDL code. On top of this functionality, any kind of analyzer tool can be implemented. Optimization algorithms can be implemented on the level of VHDL code rather than by accessing netlists. IP-Cores and Algorithms specified with vMAGIC are portable, such that they can be applied to any design.

In the following sections we present a versatile and platform-independent approach to FPGA-based testing, which makes extensive use of the vMAGIC library (available online on the vMAGIC web site). The basic principles of this approach are similar to those of Hardware-in-the-Loop (HiL) simulations, where a real Design under test (DUT) is interacting with a simulated environment. In this case, the DUT resides on an FPGA while the environment is simulated on a host computer, resulting in a high reliability of test results and, in many cases, in a speedup for the simulation itself.

3. HiLDE: A Designflow for FPGA Based Testing

Hardware-in-the-Loop Development Environment (HiLDE) is a cycle-accurate testing framework for performing FPGA-in-the-Loop simulations. HiLDE utilizes vMAGIC to encapsulate a DUT into a *hardware wrapper*, such as to enable the connection to and synchronization with a simulation tool such as MATLAB/Simulink [7], ModelSim [8] or CAMeL-View [9]. In the following, a brief description of the HiLDE wrapper and the use of vMAGIC is given, the basic concept of HiLDE has been published in [10].

There are two main challenges in the creation of HiL simulations: the synchronization of DUT and simulation on the one hand, a high-speed data transfer between DUT and simulation on the other hand. Section 3.1 gives an overview of the synchronization mechanism in HiLDE, while recent and unpublished developments in the HiLDE communication system are described in Section 3.2.

3.1. DUT Access. The HiLDE synchronization system utilizes the properties of synchronous logic to slow down the DUT execution to match the speed of the environmental simulation. This is a very special case, as typical HiL frameworks (such as [11]) must speed up the simulation to

```
VhdlFile inFile = VhdlFile.parse("test.vhd");     // parse a VHDL file
Entity inEntity = inFile.getEntity("test");       //get the input entity (”test”)
VhdlFile outFile =newVhdlFile();                  // create a new VHDL file as output
outFile.add(inEntity.clone());                    // add a copy of the entity to the out file
Architecture arch =new Architecture("beh", outFile.getEntity("test"));
outFile.add(arch);                                // create a new architecture in the output file
Component comp =newComponent(inEntity);           // create a component from test
arch.addDeclaration(comp);                        // declare the component in the out file
ComponentInstantiation inst =newComponentInstantiation("inst", comp);
 arch.addStatement(inst);                         // instantiate the component in the out file
 for(Signal s : comp.getPort().getSignals()){     // for all signals in the component
   Signal wire =newSignal(s.getIdentifier() + "_int", s.getType());
   arch.addDeclaration (wire);
   Register reg = null;
   if(s.getMode() == Signal.Mode.IN){             // create an in- or out-register
    reg =newRegister("regp_" + s.getIdentifier(), s, wire);
     inst.connect(s.getIdentifier(), wire);
   } else if(s.getMode() == Signal.Mode.OUT){
     reg =newRegister("regp_" + s.getIdentifier(), wire, s);
     inst.connect(s.getIdentifier(), wire);
   } else { /* handle other modes (INOUT, BUF, ...)*/ }
   arch.addStatement(reg);                        // and add that register to the out file
 }
 Signal rst =newSignal("LRESET_N");
 rst.setMode(Signal.Mode.IN);                     // create and add reset and clock signals
  Signal clk =newSignal("clk");
  clk.setMode(Signal.Mode.IN);
  inEntity.getPort().addSignal(rst);
  inEntity.getPort().addSignal(clk);
  System.out.println(outFile.toVhdlString());
```

Algorithm 1: Java program to generate a wrapper file for the entity which registers inputs and outputs.

real-time level, because the DUT's behavior would change at lower speeds, or it cannot be executed at a lower clock rate at all. This is especially the case when the DUT cannot be isolated from, that is, analog interfaces or other timing critical devices, such as in a production ready controller module. Under the premises of pure synchronous logic, however, the synchronization of DUT and simulation can be solved with the following interface: the hardware interface for HiLDE (see Figure 2) comprises of a bus interface [12] to the host PC, which is connected to the DUT's input and output ports, and the so-called synchronizer. The synchronizer can switch the clocks of th DUT on and off on a ”per clock cycle” basis; the user can adapt the number of clock cycles the DUT should run before synchronizing with the simulation. The integration of a DUT into a software simulator such as MATLAB/Simulink is depicted in Figure 3. The simulation itself follows four steps:

(1) read the DUTs outputs and propagate to the simulated environment (as inputs),

(2) read the environments outputs and propagate to the DUTs inputs,

(3) execute a predefined number of clock cycles (acoording to the DUTs I/O data rates),

(4) return to step 1) or end simulation.

3.2. Communication Optimization. In the simulation flow as described above, all I/O data have to be transferred at every clock cycle, resulting in redundant I/O operations where data have not changed. To decrease this overhead, two further concepts were integrated in HiLDE: *event based communication* and *transactors*:

(i) Event-Based Communication. To reduce the number of redundant I/O operations, only data that actually changes has to be transferred. While this is straightforward to be implemented in software (Simulink provides appropriate functions), the hardware wrapper has to be extended. The register of every output port is extended with a mechanism to detect changes at the output. For n_o output ports an additional register with n_o bits stores the results of these detectors, and thus indicates which values must be read by the host computer. The number of additional read operations to retrieve this information is dependent on the word width of the bus to the host computer, resulting in an overall number of read accesses $\tilde{n}_r$:

$$\tilde{n}_r = \Delta(\text{out}) + \left\lceil \frac{n_o}{\text{wordwidth}} \right\rceil, \quad (1)$$

where $\Delta(\text{out})$ is the number of output ports with a new value. Given that n_r denotes the number of read operations in the

```
ENTITY  test  IS
      PORT (
            din  :  IN STD_LOGIC_VECTOR(7  DOWNTO  0);
            dout  :  OUT STD_LOGIC_VECTOR(7  DOWNTO  0);
            LRESET_N  :  IN  std_logic ;
            clk  :  IN  std_logic
            );
END;

 ARCHITECTURE beh  OF  test  IS
     COMPONENT  test  IS
                PORT  (
                          din  :  IN  STD_LOGIC_VECTOR(7  DOWNTO  0);
                          dout  :  OUT  STD_LOGIC_VECTOR(7  DOWNTO  0)
                     );
       END COMPONENT;
       SIGNAL  din_int  :  STD_LOGIC_VECTOR(7  DOWNTO  0);
       SIGNAL  dout_int  :  STD_LOGIC_VECTOR(7  DOWNTO  0);
 BEGIN
     inst  :  test
           PORT MAP (
                          din => din_int,
                          dout => dout_int
                     );
     regp_din  :  PROCESS (clk,  LRESET_N)
     BEGIN
             IF  LRESET_N  =  '0'  THEN
                     din_int <= "00000000";
             ELSIF  clk'event  AND  clk = '1'THEN
                     din_int <= din ;
             END IF;
       END PROCESS;
       regp_dout  :  PROCESS(clk, LRESET_N)
       BEGIN
               IF  LRESET_N  =  '0'  THEN
                       dout2 <= "00000000";
               ELSIF  clk'event  AND  clk  =  '1'  THEN
                     dout <= dout_int;
               END IF;
       END PROCESS;
END;
```

Algorithm 2: VHDL output of the example Java program. The indentation and the notation of keywords is governed by the String Template file and can be changed to fit the developers needs.

standard HiLDE wrapper, the benefit $n_r/\tilde{n}_r$ is dependent on the relation of I/Os with regularly changing values to the overall number of I/Os in the DUT. In general DUTs with irregularly changing I/Os will benefit from this technique.

(ii) Transactors. Whenever the sequence of events (value changes) is predefined, such as in communication protocols, the number of I/O operations can be reduced even further by implementing adaptors for the simulation and for the FPGA. The amount of savings here is dependent on the complexity of the protocol: instead of transferring all control-signals or control-signal changes, the adaptors detect protocol activity and transfer only the necessary data, such as address and data, the actual protocol handling is processed in the adaptors in the simulation environment and in the FPGA. While the functionality of the HiL simulation is not affected by this method, the amount of I/O operations for a protocol as described in [12] can be reduced by over 90%.

3.3. HiLDE and vMAGIC. The generation of the HiLDE hardware wrapper is a very uniform procedure, usually varying only in the number and width of I/Os, or in the transfer mode as described earlier. The following steps are completed by a Java program utilizing vMAGIC:

(1) parse the DUT and a special template file,

(2) create an instance of the host communication bus and connect the synchronizer to the bus,

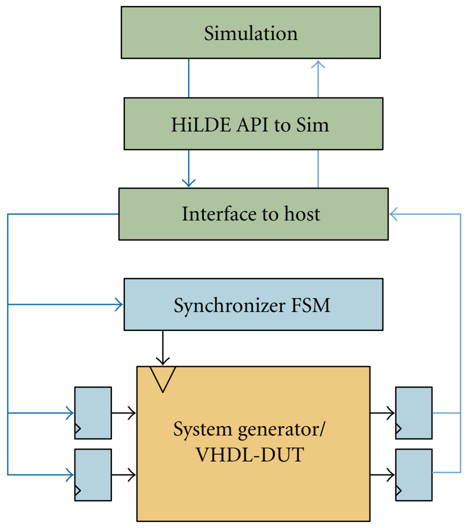


Figure 2: HiLDE hardware wrapper controlling the DUT-IOs and clock.

(3) declare and instantiate the DUT in the template,

(4) create registers for ever I/O port and connect them to the DUT instance,

(5) add all registers to the bus,

(6) generate configuration files for different simulators (currently Simulink, ModelSim and CamelView [13]).

While the manual (error prone) implementation of the wrapper can take hours, the HiLDE Wrapper Generator takes seconds at most. A demo of the generator is available at the vMAGIC project website.

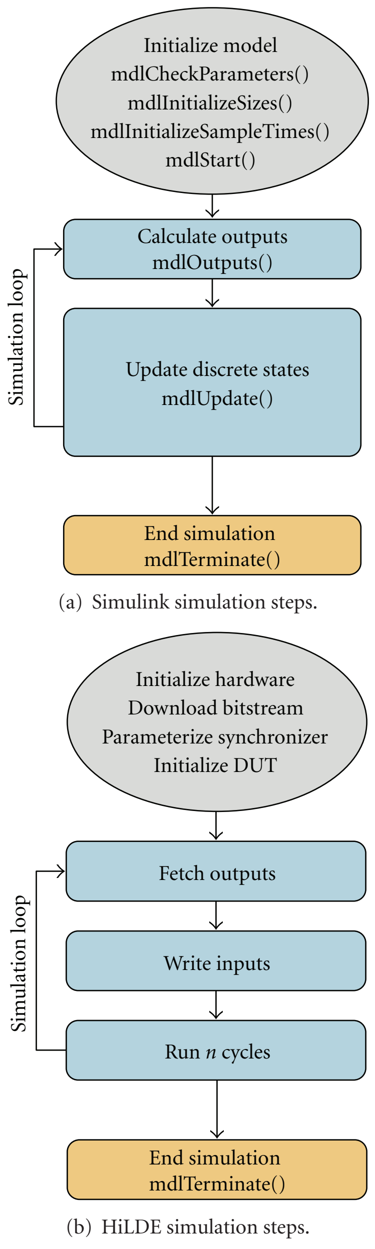
4. HiLDEGART

A logical step after performing a cycle-accurate functional design verification with a simulated environment is to realize a real-time verification of the DUT. The requirements for such a real-time framework are as follows.

(i) Monitoring of inputs and outputs of an FPGA-based system which is connected to a real testbed. To limit the data that is transferred to the host, resampling and data-based triggering of data recording must be possible, while not influencing the DUT's functionality or timing. Downsampling the data allows for a trade-off between monitoring accuracy and required communication infrastructure.

(ii) In addition to monitoring abilities, parameterization of the DUT must be possible. Switching between different parameter sets during run-time should be possible based on inputs or outputs of the DUT.

(iii) The triggering subsystem as described in what follows, should allow triggers based on boolean operations on inputs and outputs of the DUT as well as combinations of these.



(a) Simulink simulation steps.

(b) HiLDE simulation steps.

Figure 3: For Hilde simulations the standard Simulink S-Function (a) has been extended (b).

One approach is to use real-time verification tools such as ChipScope from Xilinx [14]. However, aside from its limited allowable monitoring time, this kind of tools do not permit an interaction with a DUT. Another approach is logic analyzers, which are expensive and it is very time consuming to set up a test environment. For this purpose, HiLDE for Guided Active Real-Time test (HiLDEGART) was developed. Our approach can be implemented with a standard PC, and it allows the automatic integration of an already functionally verified design to be tested in real-time. In the following section, the concept and realization of HiLDEGART is presented, focusing on the communication between the DUT and the host computer.

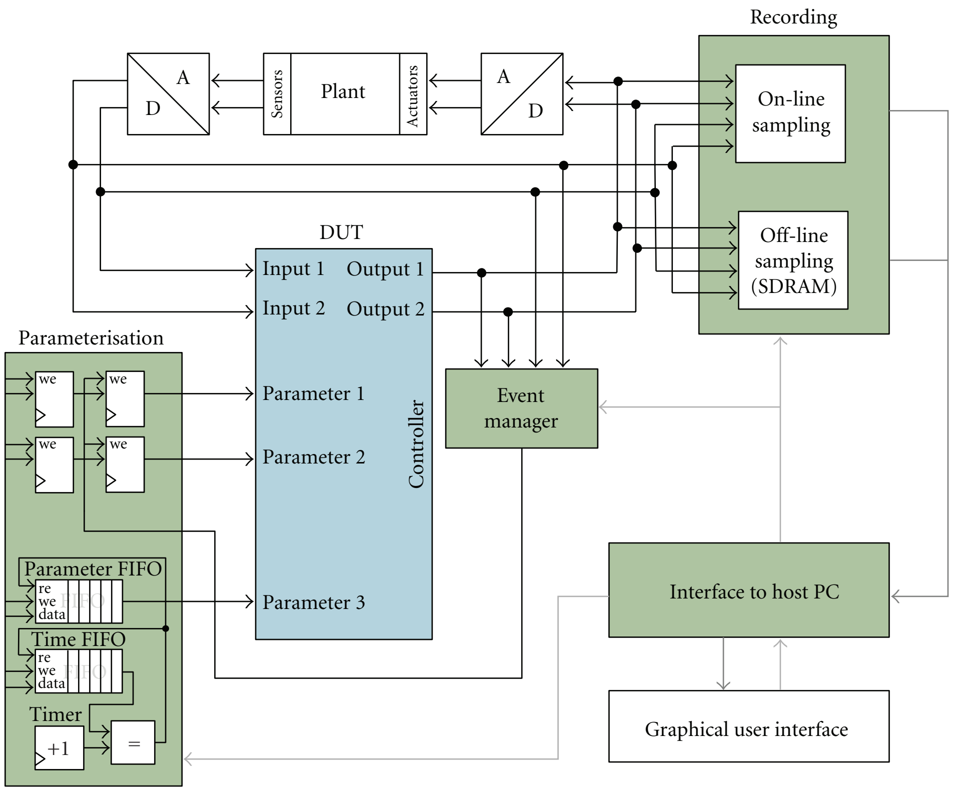


Figure 4: Structure of a real-time FPGA-in-the-Loop scenario utilizing HiLDEGART.

4.1. DUT Access. Figure 4 shows the basic concept of the presented Hardware-in-the-Loop (HiL) framework. The design under test (DUT), a controller, is implemented on an FPGA. The testbed consists of a plant to be controlled and an analog/digital interface. There are three main components surrounding a DUT to be tested with HiLDEGART.

(i) The *Interface to Host-PC* enables the communication between the host PC and the DUT. It works very similar to the interface described in Section 3.1 The main difference is the use of embedded FIFOs and external SDRAM memory to assure meeting the required sampling rates, as explained in what follows.

(ii) By utilizing the *Recording*-Block, the user has the choice to select a specific sampling rate for each port of a DUT using this module. There are two kinds of sampling mechanisms, real-time and offline sampling. Real-time sampling enables the visualisation of the selected signals at run time—the amount of signals that can be visualized in real-time is limited by external factors (e.g., I/O-bandwidth). For offline sampling, an SDRAM memory directly attached to the FPGA is used for buffering the data, allowing for very high sampling rates. The buffered data, as opposed to real-time data, is transferred to the host for visualization after the simulation has finished.

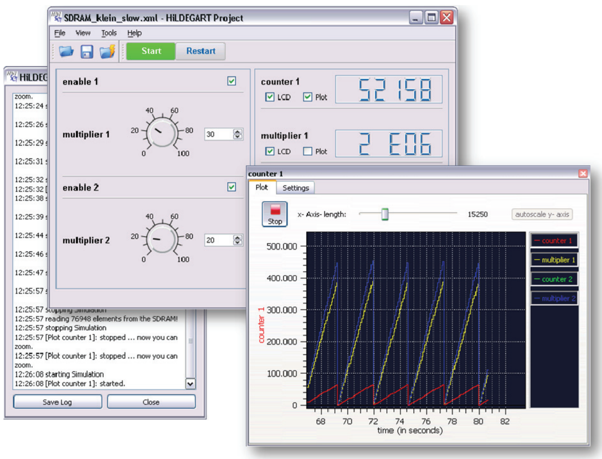
(iii) The *Event Manager* allows basic compare operations to generate events. These events may be combined by boolean operators to form conditions like $(A > \widetilde{A}) \land \neg(B = \widetilde{B})$, where A and B are the ports and $\widetilde{A}$ and $\widetilde{B}$ are values defined by the user at run time. With the resulting events, either the changing of the sampling rates or the changing of the parameters of the design can be triggered in real-time. Additionally, the events can be used to start or stop recording.

The presentation of I/O values and all configuration tasks are controlled via a GUI, which has been implemented using Trolltech's platform-independent programming environment Qt [15] in combination with QWT [16]. The project files describe the hardware interface including addresses and number representations (e.g., fix point/binary configuration). They are generated by a vMAGIC application based on user annotations (vMAGIC-tags, cf. Section 2) in the VHDL code. The GUI is automatically generated based on the interface description, including graphs, LCD-like displays for current values, and input boxes for parameters, as can be seen in Figure 5.

The automatic generation of the HiLDEGART hardware wrapper using vMAGIC is similar to the process as described in Section 3.3. The typical tool-flow for HiLDE and HiLDEGART as well as a test-case is described in the following section.

5. vMAGIC Toolflow and Example

Generating the hardware wrappers for HiLDE and HiLDEGART is an application of the complete vMAGIC designflow



```
zoom.
12:25:57 reading 76948 elements from the SDRAM!
12:25:57 stopping Simulation
12:25:57 [Plot counter 1]: stopped ... now you can
zoom.
12:25:57 [Plot counter 1]: stopped ... now you can
zoom.
12:26:08 starting Simulation
12:26:08 [Plot counter 1]: started.
```

Figure 5: Main-, Log-, and Plot-Window of HiLDEGART. The GUI is generated from an XML file generated by a vMAGIC application.

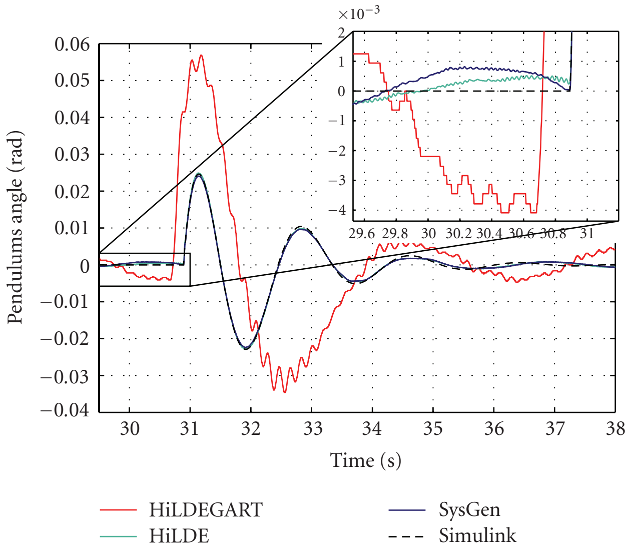


Figure 6: Inverted pendulum controller: angle of the pendulum.

as depicted in Figure 1. The starting point of the flow is a VHDL file containing the DUT's entity definition (if no internal signals should be monitored the DUT itself can be described in any HDL), which is then analyzed by vMAGIC. The user program generates the DUT-specific wrappers according to the specifications described in Section 3 respectively, 4 and generates the configuration files for a HiLDE simulation or HiLDEGART. These configuration files contain information regarding hardware addresses, sampling rates and number formats (e.g., fixpoint position). As the number of formats and sampling rates cannot be deduced from the hardware interface, they are supplied via vMAGIC-tags in the source code or directly in the GUI. This completes the vMAGIC specific part; after this, the wrapper and design files have to be synthesized using vendor specific tools. After the FPGA bitstream has been generated, the simulation is conFigurered using the configuration files and the simulation can be started.

As a case study, an inverted pendulum controller was designed using Xilinx' Simulink-based System Generator. First, a model of the pendulum and a controller to balance this pendulum are created using Simulink. The controller is then reimplemented in hardware blocks using the System Generator Toolbox. Figure 6 shows the difference between the continuous Simulink controller and the (time- and value-) discretized hardware controller (SysGen). After the System Generator model has been simulated it can be tested in real hardware using the HiLDE-flow, still using the software model of the pendulum; the result is depicted as *HiLDE*. There are very small differences between the simulation and real hardware due to the internal number representation in the system generator. The last step is to use HiLDEGART to monitor the controller in the real control loop, depicted as *HiLDEGART*. The differences between the HiLDE and the HiLDEGART simulations are due to the inaccurate modelling of the pendulum (e.g., A/D conversion effects and plant dynamics).

6. Conclusions and Outlook

In this paper, vMAGIC, a Java library for automatic code generators for VHDL has been presented. Its functionality and the associated design flow have been shown alongside with examples for vMAGIC's analysis and generation capabilities. The application areas and advantages of a vMAGIC-based designflow have been described.

The vMAGIC API has been released under LGPL 3 on sourceforge.net and can be freely downloaded and used for personal research or commercial uses. It is very usable and reliable, but by no means complete, as many useful features have not been implemented yet. We keep adding functionality to the library and we are planning to create a library on top of vMAGIC that will be able to do semantic operations as well.

Acknowledgments

This work was developed in the course of the ”Collaborative Research Center 614 - Self-Optimizing Concepts and Structures in Mechanical Engineering,” University of Paderborn, and was published on its behalf and funded by the Deutsche Forschungsgemeinschaft.

References

[1] Y. Yankova, G. Kuzmanov, K. Bertels, G. Gaydadjiev, Y. Lu, and S. Vassiliadis, “DWARV: delftworkbench automated reconfigurable VHDL generator,” in *Proceedings of the 17th International Conference on Field Programmable Logic and Applications (FPL '07)*, pp. 697–701, Amsterdam, The Netherlands, August 2007.

[2] S. McCloud, “Catapult C Synthesis-Based Design Flow: Speeding Implementation and Increasing Flexibility,” Mentor Graphics White Paper, 2004.

[3] *Synplify Users Guide*, Synopsys, Mountain View, Calif, USA, 3rd edition, 2008.

[4] *System Generator Users Guide*, Xilinx, San Jose, Calif, USA, 10th edition, 2008.

[5] C. Pohl and R. Fuest, *vMAGIC API Documentation*, Heinz Nixdorf Institute, Paderborn Germany, 2008.

[6] T. J. Parr and R. W. Quong, "ANTLR: a predicated-*LL*(*k*) parser generator," *Software: Practice and Experience*, vol. 25, no. 7, pp. 789–810, 1995.

[7] *Simulink Users Guide*, The Mathworks, Natick, Mass, USA, 2008.

[8] *ModelSim Users Guide*, Mentor Graphics, Wilsonville, Ore, USA, 6th edition, 2008.

[9] *CAMeL-View Users Guide*, iXtronics GmbH, Paderborn, Germany, 6th edition, 2008.

[10] C. Paiz, C. Pohl, and M. Porrmann, "Reconfigurable hardware in-the-loop simulations for digital control design," in *Proceedings of the 3rd International Conference on Informatics in Control, Automation and Robotics (ICINCO '06)*, pp. 39–46, Setubal, Portugal, August 2006.

[11] H. Hanselmann and F. Schutte, "Control system prototyping productionizing and testing with modern tools," in *Proceedings of the 38th International Intelligent Motion Conference*, pp. 9–16, Intertec International, Nurnberg, Germany, June 2001.

[12] H. Kalte, M. Porrmann, and U. Rückert, "A prototyping platform for dynamically reconfigurable system on chip designs," in *Proceedings of the IEEE Workshop Heterogeneous Reconfigurable Systems on Chip (SoC '02)*, Hamburg, Germany, April 2002.

[13] CAMeL-View, "CAMeL-View Virtual Engineering Workbench Reference Guide," iXtronics GmbH, Paderborn, Germany, 2004.

[14] *ChipScope Users Guide*, Xilinx, San Jose, Calif, USA, 10th edition, 2008.

[15] Trolltech, "Qt—cross-platform application framework," http://trolltech.com/.

[16] U. Rathmann, "Qwt—Qt Widgets for Technical Applications," http://qwt.sourceforge.net/.