Monitoring Executions on Reconfigurable Hardware at Model Level

Tobias Schwalb[1], Philipp Graf[2], and Klaus D. Müller-Glaser[1]

[1] Karlsruhe Institute of Technology, Institute for Information Processing Technology, Germany, {tobias.schwalb,klaus.mueller-glaser}@kit.edu
[2] FZI Forschungszentrum Informatik, Germany, graf@fzi.de

Abstract. Development, debugging and test of embedded systems get more and more complex due to increasing size and complexity of implementations. To dominate this complexity, nowadays designs are often based on models. However, while the design is on model level, the monitoring and debugging are still either on signal or on code level. This paper presents a continuous concept that allows monitoring and real-time recording of executions on reconfigurable hardware at model level. Besides developed hardware debugger modules, a development environment has been integrated. It allows on model level generation of the implementation, control of the recording and monitoring at runtime and visualization of the execution. An algorithm, running in the background, maps acquired data from the hardware to the model and commands from the model-based development environment to the hardware. The method is demonstrated using the example of statechart diagram monitoring.

Keywords: debugging, model-based control, monitoring, back annotation, real-time recording

1 Introduction

In software and hardware development costs, time-to-market and quality are often contradicting to each other, which increase with the complexity of the system. The complexity arises mainly by the increasing demands in terms of functionality, energy efficiency and the ongoing integration on hardware. This affects especially the possibilities to monitor embedded systems at runtime. To dominate the increasing complexity, more and more abstract approaches for the development of embedded systems come up. One option is model-based development using graphical languages, for example Unified Modeling Language (UML)[1], statecharts[2] or signal flow graphs[3]. These models can be used to automatically generate source code for programming embedded systems.

To preserve the benefits of flexible development and simultaneously achieve high performance, reconfigurable hardware devices (e.g. Field Programmable Gate Arrays - FPGAs) are deployed. These offer the possibilities to implement algorithms in hardware and parallel execution, to gain more computing power. However, developing reconfigurable systems is more complex, especially in the area of debugging and testing with regard to real-time conditions.

The challenge is to combine these domains into a continuous model-based development process that allows debugging and monitoring on model level [4]. In a previous paper [5] an initial concept for model-based debugging of reconfigurable hardware has been presented. The paper focused mainly on the overall concept and the on-chip hardware architecture for real-time recording. This paper presents an advancement of the concept and focuses on the control of the functionality at runtime and the mapping between the different abstraction levels. In this context, the next chapter describes the state of the art in terms of monitoring and debugging reconfigurable hardware. In Section 3 the concept allowing model-based debugging on reconfigurable hardware is described, while Section 4 gives a brief overview of the on-chip architecture. The following section focuses on the software used for instantiation and runtime control. Section 6 introduces the method to achieve a mapping between hardware implementation and model to allow back annotation to model level and control of the debugging. The next section shows carried out tests and their results. We close with conclusions and outlook on future work in Section 8.

2 State of the Art

2.1 Simulation vs. Debugging

In the FPGA development process simulation[6] and debugging[7] are used for the identification of errors. Simulation, compared to debugging, has the advantage that a hardware system is not necessary, therefore it can be performed earlier in the development process. Using simulation, all signals of the design are directly accessible and their behavior can be displayed. Using debugging, the access to internal signals is limited, because the signals need to be either recorded on-chip (limited memory) or forwarded to output pins (limited number) for external processing. In general, an embedded system exists in context of its peripherals and surroundings. In a simulation all these need to be additionally integrated and it is very complex to consider all parameters. Therefore, there is always an uncertainty if the simulation represents the real system. With debugging, peripherals and surroundings are present in the real system and do not need to be simulated. In addition, using simulations it is difficult to determine non-functional parameters, for example real-time conditions or performance.

2.2 Debugging Reconfigurable Hardware

For debugging FPGAs [8] it has to be mainly distinguished between real-time and non real-time debugging. In the latter, breakpoints[3] stop the clock of the design under test or it is executed step by step. When the design stops, the status of the on-chip registers is read back and interpreted to determine the system state. Disadvantages are that performance and timing cannot be analyzed and it is not possible to obtain the status of signals before stopping the system. For real-time

[3] Configurable event triggered by a set of conditions that halt the system

debugging essentially two different methods exist: the first forwards the signals of interest to output pins of the FPGA. Recording and processing is performed by external hardware (e.g. digital logic analyzer). The number of signals is limited, because every signal requires an extra output pin. The advantage is that on-chip logic or memory is not needed. The second option integrates additional on-chip modules to record the signals and transfer the data via an interface to a PC. An example is ChipScope[9] by Xilinx, it stores the signal flow in on-chip Block-RAM and transfers data via the JTAG interface. This method can record many signals, but recording time is limited by on-chip memory.

All discussed debuggers work and get controlled during runtime on signal or code level, i.e. breakpoints or trigger conditions for recording are set with respect to the signals in the design. This it is not suitable for a model-based design process, because the developer designs the system on model level and does not know about signals or source code, as this is mostly automatically generated according to the developed model.

2.3 Model-based Debugging

Debugging on model level for embedded systems is possible, but not widely used. The commercial software Matlab[10] offers model-based debugging in their Stateflow part, it allows the implementation and debugging of statechart diagrams, but supports debugging only on special microprocessor platforms.

A general approach for model-based debugging on embedded systems has been presented in [11], [12], [13]. These papers describe a concept which refers to different abstraction layers in a model-based design process and a framework for a modular system architecture. Also a prototype implementation and a connection to a real-time in-circuit emulator are shown. In this paper, these principles are extended with real-time aspects within reconfigurable hardware and the automatic generation of the hardware debugging platform. In this context, a mapping between model and hardware platform is developed, that additionally allows the control of the debugging during runtime from model level. The concepts of the presented debugging environment [13] are integrated in a new developing environment that is based on model-based developing frameworks and extended with interactive control modules.

3 Model-based Design Flow for Debugging

The development of reconfigurable systems is heading towards a model-based design flow. Hence also monitoring, debugging and control of the debugging needs to take place on model level. A continuous concept for debugging on model level is depicted in Figure 1.

The flow shows on the left side the design and transformation, in the middle bottom the execution and on the right side the mapping and debugging. In the middle, a direct connection for model-based control is added. In the beginning, the user designs his system using different models. In this context, the models

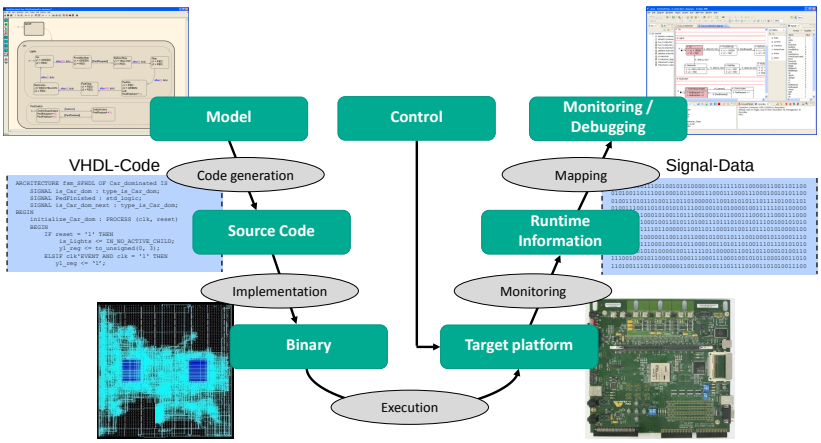

Fig. 1. Design flow for model-based monitoring and control [10],[14],[16]

have to be sufficiently detailed to generate executable source code. This automatic code generation is currently supported by various toolsets and mainly allows the generation of C code and partly HDL code. As we examine reconfigurable hardware, we concentrate on VHDL code. To enable debugging, the generation process needs to implement an interface for debugging into the hardware design. After generation, the design is synthesized by the FPGA-specific tools using mapping and routing algorithms to generate a bitstream, which is used to program the FPGA. The design, which includes the user implementation and additional debugger modules, is executed on a FPGA. During execution in the right branch of the process the signals of interest are captured by the debugger modules. The data is transferred to a PC during monitoring or after recording. Since the data obtained on-chip relates to signal level, it is mapped to the model. In visualization the user can monitor the execution.

During the whole process, the user is working on model level, the intermediate steps are performed by algorithms. Therefore, developing, debugging and control of the debugging take place on the same abstraction level. The automatic interpretation and mapping of signals to the model cannot be regarded as reverse engineering, since information from the left branch of the process is needed. It is rather a reversal of the transformation from model to hardware.

4 On-chip Architecture

The modular architecture of the debugger modules is depicted in Figure 2. First the recorded signals of the designs under test are buffered in a FIFO. This first FIFO can also run in a ring buffer mode, which allows recording signals before a trigger is released. This mode allows easier identification of the reason of an error, because the history of events can be recorded and parts of the system state reconstructed. After the FIFO, the data can be processed by different

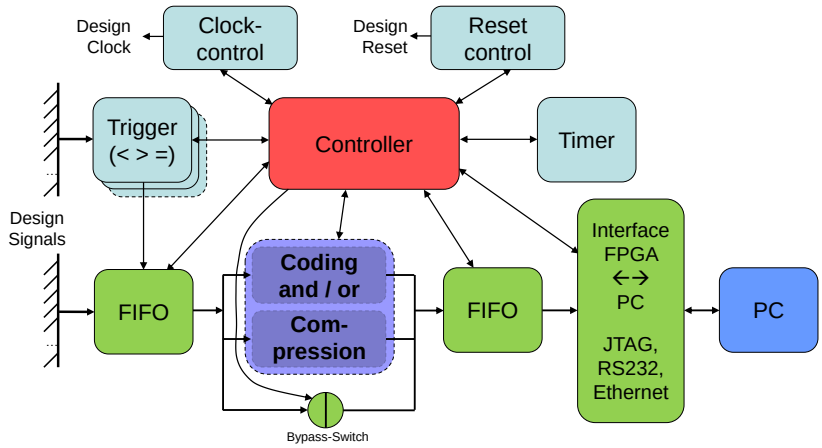

Fig. 2. Architecture of the on-chip debugger modules

compression or coding algorithms to get a reduction of the data. This processing is optional, but allows to use on-chip memory more efficiently and record a longer period of time. The second FIFO stores the data into on-chip memory, before it is transmitted to a PC using a communication interface. This architecture is described in more detail in [5]. In comparison to the original design, the Pseudo-Random-Generator is excluded and the coding and compression unit is shirked to decrease the use of logic resources. The DDR-Interface is no longer supported, because of its speed in comparison with internal memory in the actual system.

Besides recording, a direct monitoring of the design signals is also possible, since the transmission is independent of the recording. In this mode compression is bypassed and the FIFOs are directly read out. However, a restriction is the bandwidth, which depends on the speed of the interface to the PC and the processing in the model-based development environment. Therefore, real-time monitoring is only possible within small systems with few signal changes.

The debugger is managed by the controller, which is a small microprocessor. It monitors the status, controls recording and the design under test as well as communicates with the PC. To control the design under test, its *clock enable* and *reset* signal can be changed, which allows to stop and reset the design independent of the debugger modules. As inputs, trigger modules monitor signals of the design under test on the occurrence of certain conditions to start or stop the recording. The modules can trigger on edges or conditions of signals as well as compare signals to other signals or with fixed values. The trigger module can be repeatedly instantiated to allow complex chained comparisons. Additionally, the trigger conditions can be modified at runtime, switching integrated multiplexers or changing memory cells. The controller also communicates with the PC, receiving commands for control and transmitting recorded data. The design has been extended by a timer module to enable time-based recordings.

5 Software and Model-based Control

For model-based generation of the platform, visualization of the execution and control of the debugging at runtime a model-based developing environment has been implemented. The environment is build on Eclipse[14], extended with the Eclipse Modeling Framework (EMF) and Graphical Modeling Framework (GMF) for model-based design and the xPand framework for code generation.

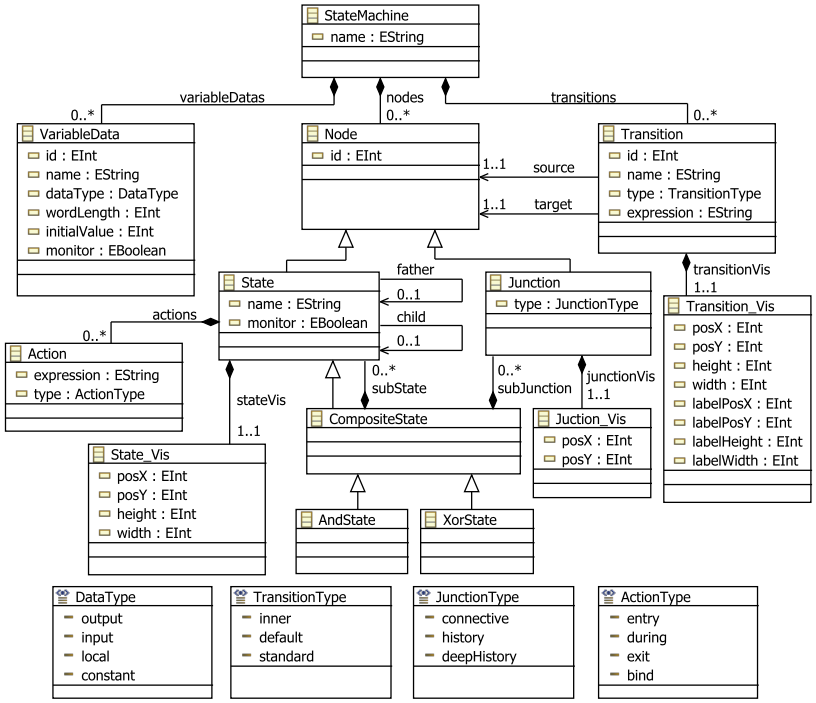

Fig. 3. Meta model for statechart diagrams with monitoring extensions

With regard to statecharts in the first step, a meta model has been developed (Figure 3). As we use Eclipse the meta model is based on the Ecore meta meta model. The meta model relies on the design of statecharts in Matlab Stateflow to get the opportunity to convert Matlab files into the developed IDE. It is similar to the UML statechart meta model [1], but simplified concerning states and transitions. The additional class *VariableData* keeps the inputs, outputs and internal variables that are used within the statechart for communication. All main classes, namely *Node*, *Transition* and *VariableData* have an attribute *id*, which allows direct identification of the individual element. The classes on the bottom show enum-classes defining different types of elements within the meta model. The additional visualization classes (*..._Vis*) store layout information, if

a Matlab Stateflow model is converted. The flag *monitor* integrated in the class *State* and *VariableData* is used during the generation of the platform to specify the monitored instances.

According to the meta model, three models are created in the GMF-framework concerning the graphical editor on model level. The first model describes the palette in the editor, i.e. the tools that are available to modify the model. In the example, there are tools to draw simple states, xor-states, and-states and junctions as well as transitions between states. The second model, the gmfgraph model, describes the graphical representation of the elements in the model, i.e. their shape, color etc. The last model layouts a mapping between the three models, it creates a connection between elements in meta model, tools and graphical representations. After creation of these models a model-based IDE can be generated by the framework. The result is depicted in Figure 4 (middle and left part). In the middle, is the modeling area with the tool palette and on the left side is the project management. The windows on the bottom and on the right are additionally implemented and explained in the next paragraph.

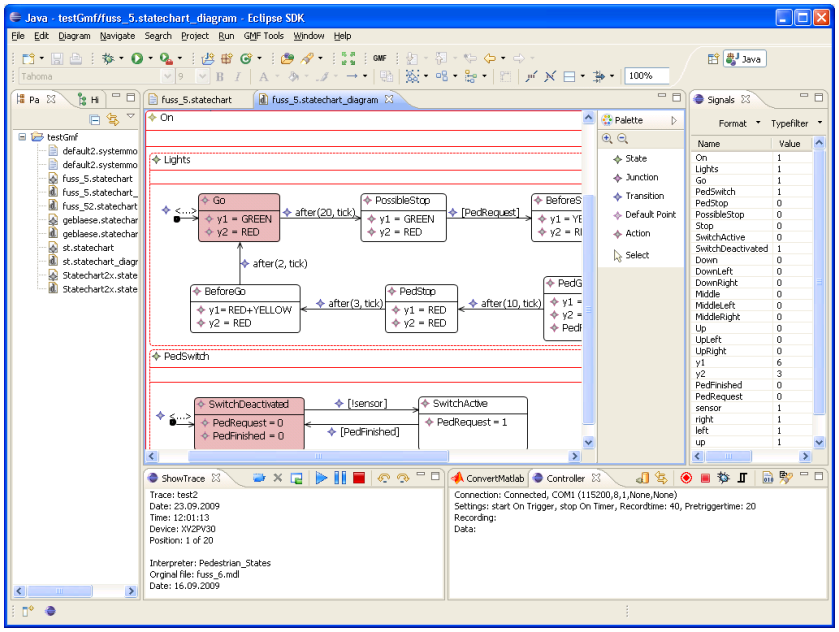

Fig. 4. Model-based development environment

The window on the right allows displaying all recorded data with regard to the model. It displays in addition to the active states, which are shown in the model, the monitored values of internal variables, inputs and outputs. The view in the middle below can be used to visualize executions that have already been recorded and saved to a file. The window on the right below is the controller

for on-chip recording and monitoring. It integrates a connection to the on-chip debugger to send commands and receive data during runtime.

The controller window allows controlling the *reset* and *clock enable* signal of the design under test. To control the recording start- and stop-conditions can be specified. The recording can start on the trigger, with the start of the design or manually. However, in this context, a manual start does not fulfill real-time conditions, because the time between the click and the actual start of the recording in the system cannot be exactly determined. The recording can stop either on a second trigger, on a timer or when the recording memory is full. Using the timer the recording time can be specified with regard to the number of clock cycles. Also the pretrigger time can be specified the same way.

Additionally, the trigger conditions are specified in the controller window. These are described with regard to the model, for example the string *in(Go) and in(SwitchActive)* would specify the trigger to release when state *Go* and *SwitchActive* are active at the same time. The trigger can be specified according to states, inputs, outputs and internal variables. The complexity have to match the implemented number of trigger modules, i.e. if the trigger condition is compound from three statements also minimal three trigger modules have to be present in the hardware implementation. In the next step, the design under test can be started, i.e. the *clock enable* signal is released and/or a *reset* performed. When recording is finished the data is transferred and stored in a XML file, which can be directly visualized to perform a postmortem analysis[4]. This enables model-based real-time debugging, but as the parameters have to be setup before recording, some knowledge according to the error needs to be present.

In another option, the controller can directly monitor the execution on-chip. The status of all recorded signals are polled every 100ms and the transmitted data is directly interpreted and visualized. The clock in the design under test can run continuously or controlled step by step to enable slow execution. No real-time debugging is possible using direct monitoring, because it only allows either slow execution (step by step) or slow monitoring (every 100ms).

6 Mapping of Model and Hardware Implementation

In a model-based development process the design of the system is on model level and the source code for programming is mostly automatically generated from the model. If there is an error in the system, the user wants to monitor and debug his system on model level - the same level it has been designed. However, the data on a FPGA relies on signal level, therefore a mapping between model and hardware is needed. In addition, in a FPGA internal signals representing model elements are not directly accessible, therefore during generation of the system an adapted debugging interface needs to be integrated.

The design flow for generation of the platform and mapping of model and hardware system is depicted in Figure 5. In the example Matlab Stateflow is

[4] Analysis that is performed after an expected event (e.g. an error / a system crash)

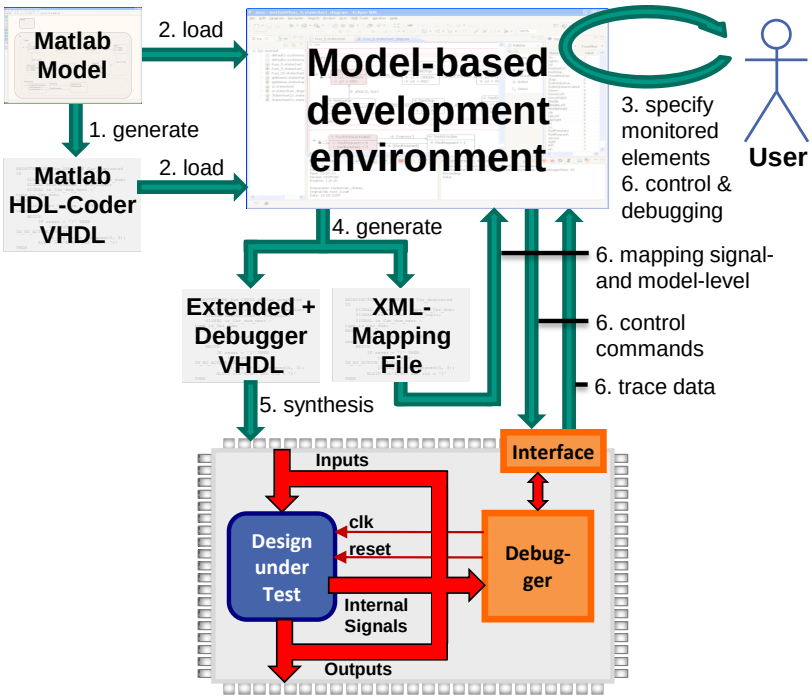

Fig. 5. Flow for platform generation and mapping of model and hardware

used to design the statechart diagram and the Matlab HDL-Coder to generate the VHDL code, which represents the functionality of the statechart. If the model is designed and the VHDL code generated, both is load into the described development environment (see Section 5). The Matlab file is converted to an EMF model file which is based on the meta model shown in Figure 3. For efficient use of the FPGA resources in the next step the user can specify in the model the monitored elements.

According to the specifications in the model, the (by Matlab) generated VHDL code is extended to enable monitoring of inner states and variables. Transitions cannot be monitored directly, but as they form connections between states, according to changes from one state to the next, the used transition can be determined. The monitored signals are grouped together into two vectors, one for recording and a second for the trigger, and integrated as outputs in the VHDL code. The debugger modules are connected to the statechart module by a generated interface file. The interface is a top level VHDL structural description and connects signals from outside to signals in the design and specifies signals between the debugger modules and the statechart module. As all names of the signals are known or read from the Matlab file, the interface file can be directly generated.

To get the mapping between signals and types in the VHDL file and elements in the model, an algorithm has been developed. This algorithm uses the fact, that in the VHDL code all signals and types have the same name as in the model and that all states in a composite state are grouped together. Therefore, as the model and VHDL always follow the same principles the mapping can be identified. This mapping is stored in a XML file, because it is needed later during debugging. The file contains the name, position, size and the function of the signals in the vectors with regard to the elements in the model. The file also contains general information concerning the model, which later allows later an easier identification.

The generated VDHL code is in the next step synthesized by FPGA specific tools to generate a bitstream, which is integrated on the hardware. When the design is executed the debugger modules record the specified signals. The debugger modules are independent of the model and all signals, as described, are forwarded to the debugger in a vector. Therefore, during debugging additional information is needed for visualization of the execution on model level and control of the debugging (e.g. setting the trigger conditions). This information is contained in the XML file generated in the previous step. Therefore, the user can debug the system on model level and control the debugging according to the model notation. The back annotation of the recorded signals to the model and generation of control commands is performed by algorithms in the background using the mapping information.

In general, the back annotation from hardware to the model follows the same principles as the back annotation in a general software debugger [15], after transmitting the data gained on low level, it is combined with the mapping information to get a representation on high level. However, with respect to reconfigurable hardware it needs to be regard that processes can run in parallel and that a single element in the model can be represented by many signals. Also the coding and interpretation of the signals (binary, high-active, low-active, ...) according to the status of the elements in the model needs to be considered.

7 Integration and Test

Different tests have been carried out to evaluate the functionality and integrity of the depicted method and platform. The tests are mainly performed using a development board, including a Xilinx Virtex II Pro FPGA [16] as well as interfaces for communication and programming. The size of the debugger modules is variable and depends on the number of signals for recording and triggering as well as the possible recording length. It also depends on the number of trigger modules (i.e. the possible complexity of the trigger condition) and the type of compression unit.

Different designs were evaluated from small models used with minimal debugger modules up to large systems recording 128 signals. A common example is described in more detail. It is based on a model, which describes the traffic light system at a crosswalk and includes 14 states, 1 input, 2 outputs and 2 internal

variables. The model is designed in Matlab Stateflow and according VHDL code generated. Both is load in the model-based development environment (see model in Figure 4). In the example every element in the model is selected for recording. After specification the according VHDL and XML files are generated. The VHDL design is integrated on the FPGA using the Xilinx ISE design suite, some additional adjustments are carried out according to the FPGA, the clock signal and external connections. The debugger modules record altogether 32 signals at 100MHz with a depth of 1024 clock cycles. Therefore, the recording data rate is 3.2Gbit/s. After recording or during monitoring the data is transferred to the PC using a RS232 interface. In the example the debugger modules use approximately 4% of the FPGA resources. The debugger does not include coding or compression, which would significantly increase FPGA resources.

During debugging the development environment connects to the debugger modules on the FPGA using the integrated RS232 interface. The XML file provides a mapping between the hardware implementation and the model and allows the user to control the debugging according to the crosswalk model. Further, the XML file is used to visualize the internal execution of the system in the model, highlighting active states and displaying the value of inputs, outputs and internal variables. Besides the real-time recording and postmortem analysis the system could be directly monitored during runtime. The transmission, interpretation and visualization of the status of 32 signals in the design every 100ms are performed without any timing problems. However, this is only a data rate of 320bit/s, therefore there is no real-time monitoring possible. In another design - with 128 signals - the transmitting interval even needed to be reduced to 250ms to process the data before the next is transmitted, the most of that time thereby is consumed by the graphical visualization.

8 Conclusion and Outlook

A method for model-based real-time recording and monitoring on reconfigurable hardware has been presented. Therefore, the possibilities of an abstract and complex functional and algorithmic inspection of reconfigurable system have been increased. In comparison with present techniques, the user does not only develop on model level, but can also debug and monitor as well as control the debugging on the same level. All intermediate steps from the model to hardware implementation and vice versa are carried out automatically by algorithms.

The underlying hardware modules are capable of monitoring and real-time recording as well as independent of the reviewed model. The debugger provides high modularity and adjustability during runtime, allowing several ways to identify causes of errors without re-synthesizing the design. The integrated development environment allows automatic integration of the debugger using a generated interface, control of the debugging during runtime and visualization of the execution - all on model level. The algorithms in the background convert the received data from hardware to model level and convert the commands from model level to hardware.

In future, there will still be many developed modules on code level, therefore the concept could be extended to allow mixed debugging on model, code and signal level. According to the software, the integration of breakpoints on model level could be added allowing more specific debugging scenarios. Also the configuration of the debugger in terms of the compression and complexity of the trigger condition, which is at the moment performed in the VHDL code, could be integrated into software. In addition, the IDE will be extended to allow complete code generation of statechart diagrams to get independent of Matlab.

References

1. Object Management Group: Unified Modeling Language (UML) Specification, Version 2.2 (2008)
2. Harel, D.: Statecharts: A Visual Formalism for Complex Systems. Science of Computer Programming 8 (1987, 3), pp. 231–274 (1987)
3. Barry, J. R., Lee, E. A., Messerschmitt, D. G., Lee, E. A.: Digital communication. Springer, New York (2004)
4. Schmidt, D.C.: Model-Driven Engineering. J. Computer. Vol. 39 Iss. 2, pp. 25–31 (2006)
5. Schwalb, T., Graf, P., Müller-Glaser, K.D.: Architektur für das echtzeitfähige Debugging ausführbarer Modelle auf rekonfigurierbarer Hardware. In: Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen, pp. 127–137. Berlin (2009)
6. Howe H.: Pre- and postsynthesis simulation mismatches. In: Verilog HDL Conference, pp. 24-31. IEEE International, Santa Clare (1997)
7. Lach, J., Mangione-Smith, W., Potkonjak, M.: Efficient error detection, localization, and correction for fpga-based debugging. In: 37th Design Automation Conference, pp. 207-212. Los Angeles (2000)
8. McKay, N., Singh, S.: Debugging techniques for dynamically reconfigurable hardware. In: Seventh Annual IEEE Symposium on Field-Programmable Custom Computing Machines, pp. 144–122. Napa Valley (1999)
9. Arshak, K., Jafer, E., Ibala C.: Testing fpga based digital system using xilinx chipscope logic analyzer. In: 29th International Spring Seminar on Electronics Technology, pp. 355-360. ISSE06, St. Marienthal (2006)
10. Mathworks: Matlab & Simulink (2010) `http://www.mathworks.de/`
11. Graf, P., Hübner, M., Müller-Glaser, K.D., Becker, J.: A Graphical Model-Level Debugger for Heterogenous Reconfigurable Architectures. In: 17th International Conference on Field Programmable Logic and Applications (FPL), pp. 722–725. Amsterdam (2007)
12. Graf, P., Reichmann, C., Müller-Glaser, K.D.: Towards a Platform for Debugging Executed UML-Models in Embedded Systems. In: UML Modelling Languages and applications, pp. 238–241. Springer, Heidelberg (2004)
13. Graf, P., Müller-Glaser, K.D.: ModelScope Inspecting Executable Models during Run-time. In: 30th International Conference on Software Engineering, pp. 935–936. Leipzig (2008)
14. Eclipse Foundation: Eclipse Modeling Project (2010) `http://www.eclipse.org/modeling`
15. Rosenberg, J.B.: How Debuggers Work. John Wiley & Sons Inc., New York (1996)
16. Xlilinx: Virtex-II Pro and Virtex-II Pro X - FPGA User Guide v.4.2 (2007)