Visualized SystemC Debugging

Christian Genz[1], Frank Rogin[2], Rolf Drechsler[1], Steffen Rülke[2]

[1] {genz,drechsle}@informatik.uni-bremen.de

Institute of Computer Science
University of Bremen
28359 Bremen, Germany
http://www.informatik.uni-bremen.de/agra

[2] {frank.rogin,steffen.ruelke}@eas.iis.fraunhofer.de

Design Automation Division
Fraunhofer Institute for Integrated Circuits
01069 Dresden, Germany
http://www.eas.iis.fraunhofer.de

Abstract

We present an integrated debugging environment that facilitates designers in simulating, debugging, and visualizing their SystemC models combining high-level debugging with a visualization frontend.

1. Introduction

SystemC is a widespread system level description language that facilitates system architects to specify their designs using a larger spectrum of abstraction levels than traditional Hardware Description Languages, like VHDL or Verilog, do. There, various abstraction levels, different involved components (IP, SW/HW), and diverse tools further complicate design comprehension. Several sources show that verifying a complex design requires often more than 50% of the overall design time. Unfortunately, the amount of functionality given by SystemC increases the effort in observing simulation results, and in debugging an erroneous design. Currently SystemC does not comprise sophisticated debugging aspects, nor it provides any visualization support. Furthermore, language features such as multithreading and event-based communication increase the program complexity and introduce nondeterminism in the system behavior. Thus, debugging SystemC models is a challenging task.

In this work we introduce an integrated debugging environment (IDE) for SystemC. Besides simulation control and data hiding our approach extends the data introspection capabilities of SystemC. It is non-intrusive and does not alter the simulated model, nor the simulation kernel, or additional libraries (C++ STL, SCV). Our solution combines high-level SystemC debugging [1] with a visualization frontend [2]. Thus, the user debugs and visualizes a SystemC model at a high level of abstraction working with signals, ports, events, and processes. The debugger kernel is based on the Open Source debugger GDB[1] while the visualization component makes use of the visualization engine from Concept Engineering GmbH[2]. It generates different views of the model, supporting cross-probing and annotation of the visualized context. Our environment runs under Linux/Unix. During a debug session the user has various possibilities to present dynamic and static debugging information by the visualization frontend or in the debugger console window. Thus, he gets a fast and concise insight into the observed SystemC model which accelerates and eases defect detection, understanding, and location.

[1] www.gnu.org/software/gdb
[2] www.concept.de

2. General architecture

The IDE consists of three components. Each of these components realizes a different task. As sketched in Figure 1 our debugging flow starts at the original system description which is being compiled to an executable. The executable can be run in the debugger. In parallel the system description is statically analyzed by the visualizer. The intermediate representation (IR) that is generated after analysis can be used to render the model inside the graphical frontend. RTLVision from Concept Engineering is used for this purpose.

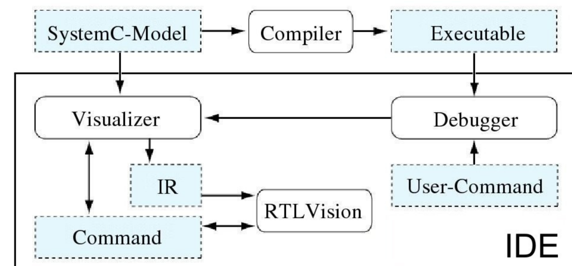

Figure 1: Architecture of the IDE

After passing the SystemC evaluation phase successfully the debugger waits for user commands. Those commands can be used to show or to hide details inside the visualization frontend, as well as to control the simulation of the executed model. All commands that influence the graphical view are directly propagated to the visualizer. Being aware of the model structure the visualizer assembles commands and maps SystemC components to the appropriate graphical symbols. Thus, RTLVision can be instructed to switch to specific parts of the design and to update signal values during execution.

3. Debugging features

To illustrate the utilization of our IDE we used the RISC-CPU design that is provided by the OSCI SystemC v2.0.1 library package[3]. Figure 2 shows an example debug session

simulating this design. Here, different views allow to explore a design at various abstraction levels. Static and dynamic debugging information are presented either by different colorings, info boxes, labels, and dedicated component displays in the GUI, or as text output in the debugger console.

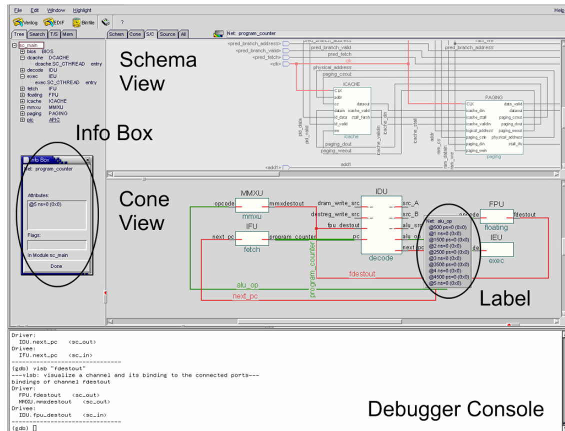

Figure 2: Example debug session

Besides standard C++ and high-level SystemC debugging features the IDE provides several commands which interactively control the visualization of a SystemC design and its simulation state. Two command types can be distinguished:

- **Examining commands**. These commands allow getting a fast insight into the parts of a design relevant for the actual debug session while non-relevant data are explicitly excluded.
- **Observation commands**. Commands of this type support the user in obtaining different data about the simulation state (such as signal/port values, or process activations) logged over/at a specified simulation time.

Next, two commands illustrate the provided visualization functionality exemplarily.

The **vlsb** command visualizes the specified channel and all connected modules in RTLVision. In case of a failure related to a specific signal the user gets a quick overview about all its connections. Thus, he can focus his error search to the relevant modules only which helps accelerating debugging. Figure 3 sketches the **vlsb** visualization output after calling it with two signals of the RISC-CPU design:

```
(gdb) vlsb "ram_cs"
(gdb) vlsb "next_pc"
```

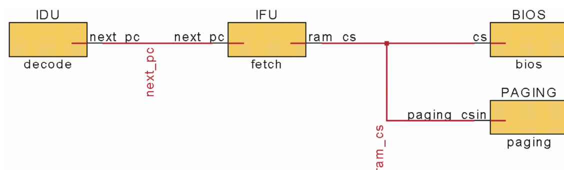

Figure 3: Example command **vlsb**

The **vtrace_at** command is a typical representative of the observation command type. It traces the given signal or port and records the actual value at the specified simulation time stamp. The logged value is attached as label text in RTLVision and can be displayed in an info box additionally. Logging dedicated signal values during simulation is very helpful when the user does not exactly know what is going wrong and when the defect infection occurs. Figure 4 illustrates the visualized tracing of the top-level signal *addr* in the RISC-CPU design at different time stamps to check whether the right addresses are forwarded to the RAM:

```
(gdb) vtrace_at "addr" 42000
(gdb) vtrace_at "addr" 46000
(gdb) vtrace_at "addr" 50000
(gdb) c
 ...
(gdb) vlsb "addr"
```

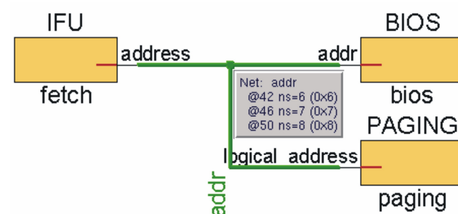

Figure 4: Example command **vtrace_at**

4. Conclusion

In this work we introduced an approach for SystemC debugging, based on a visual interface and the GDB debugger. We demonstrated the advantages of our visualized debugging commands while applying them to the RISC-CPU design of the SystemC library. Future work will provide more sophisticated debugging functionality and will include a tighter coupling between the visualization frontend and the debugger.

5. References

[1] F. Rogin, E. Fehlauer, S. Ohnewald, S. Rülke, and T. Berndt, “Non-intrusive High-level SystemC Debugging“, *In Forum on Specification and Design Languages*, 2006.

[2] C. Genz, R. Drechsler, G. Angst, and L. Linhard, “Visualization of SystemC Designs”, *In Proceedings of ISCAS*, 2007.

Acknowledgement

The authors would like to thank Lothar Linhard and Gerhard Angst from Concept Engineering for their friendly support.

[3] www.systemc.org