Non-Intrusive High-level SystemC Debugging*

Frank Rogin[1], Erhard Fehlauer[1], Steffen Rülke[1], Sebastian Ohnewald[2], Thomas Berndt[2]

[1] Fraunhofer IIS / EAS
01069 Dresden, Germany
{rogin,fehlauer,ruelke}@eas.iis.fraunhofer.de

[2] AMD / Dresden Design Center (DDC)
01109 Dresden, Germany
sebastian.ohnewald@amd.com
thomas.berndt@amd.com

Abstract

We present a non-intrusive high-level SystemC debugging approach to be used with SystemC v2.0.1 and GNU debugger gdb. Our approach is integrated into an industrial design flow and enables developers to debug designs at high level working with signals, ports, events, and processes. Thus, one gets quick and concise insight into static structure and dynamic behavior of the design without the burden of gaining detailed knowledge of the underlying SystemC simulation kernel. Only minor transparent changes to SystemC kernel source code are required, whereas there is no need to touch the flow or the designs. Practical experiences show promising results.

1 Introduction

System level design methodologies promise to address major challenges in modern System-on-Chip (SoC) designs. System level design embraces various abstraction levels, different components (IP, SW/HW), diverse tools, and methodologies which further complicate design comprehension. Studies revealed that today often more than 50% of design time is spent to verify a complex design that means to identify, understand, localize, and correct errors [Che03].

Many system level languages and design environments were proposed over the last years, e.g. [BWH03, GZD00]. One of the most popular languages of this type is SystemC [SCI06]. It has become a de facto standard in industry and in the academic field. SystemC provides concepts such as object-orientation, concurrency, and high-level modelling.

* This work is co-funded by the SAB project Sauber.

Currently SystemC does not comprise debugging aspects. It solely defines functions to trace module level variables and signals. Traced values are written to file during simulation, and analyzed with standard tools afterwards. Standard C++ debuggers are applied to analyze a functions local variables during simulation run. Unfortunately, both debugging approaches operate on very low abstraction level. Especially, standard C++ debuggers do not understand specific SystemC constructs. Besides, SystemC maps modules onto individual threads of execution which leads to non-linear execution sequences. This makes predicting which module will be active next extremely difficult.

As working at appropriate abstraction levels is an essential means to understand designs and to fix bugs quickly, several commercial and research tools have been developed dealing with high-level SystemC debugging. Some of the available commercial solutions and academic prototypes are listed and assessed below.

MaxSim Developer Suite [ARM06] comprises a block level editor, and simulation, debugging and analysis tools. It addresses architectural analysis as well as SystemC component debugging at low level and at transactional level. *ConvergenSC System Verifier* [SVH06] targets SystemC system level design and verification. It utilizes a simulation kernel which is specially adopted to fit SystemC needs. Its integrated debugger offers SystemC specific commands supporting breakpoints and SystemC QThreads at source level. *CoCentric System Studio* [Syn06] supports SystemC design, simulation and analysis at system level, and partly synthesis from behavioral and RT level. It utilizes standard C++ debuggers (e.g. gdb), i.e. it does not handle SystemC constructs in a specific way.

[GDL03] presents a method to extract structural data from SystemC designs automatically, and to pass it to a commercial visualization tool using an application programming inferface (API). The SystemC kernel has been modified to interface to the API. [Gra06] uses SystemC simulation results to create Message Sequence Charts to visualize SystemC process interaction at a high level. Filters cut out parts of inter-process communication in order to reduce information complexity. [CRA01] applies the observer pattern [GHJ99] to connect external software to the SystemC simulation kernel. This general method facilitates loose coupling and requires just minimal modifications of the SystemC kernel.

None of the tools mentioned above fully meets the requirements to integrate high-level SystemC debugging into the existing design flow at AMD, namely to

- fit seamlessly into the flow where designers used to apply gdb either at command line or through a GUI,
- easily access static and run-time design information at system level,
- work with an existing SystemC kernel,
- avoid changes to the design to support debugging, and
- exercise high-level breakpoints.

So we decided to implement high-level SystemC debugging as a set of gdb user commands to avoid patching of gdb source code. C++ routines and shell scripts collect required data from SystemC or gdb runtime, respectively, and present the information to the designer. Only minor transparent changes to SystemC kernel source code were made to enhance debugging performance.

The remainder of this paper is organized as follows. Section 2 proposes our high-level SystemC debug methodology derived from the given industrial requirements. Section 3 introduces implementation details, and explains modifications made to improve performance. Section 4 presents some practical experiences gained. And section 5 concludes the paper.

2 Methodology

2.1 Requirements and design issues

The most important industrial requirement was the demand for a non-intrusive debugging facility that fits seamlessly into the existing design flow. That means, the solution should work with the available SystemC kernel and avoid any changes to present designs or (third-party) IP blocks. On the tool side, the already applied GNU debugger gdb [GDB06] should be extended without any need for patching its sources. Advantages are a familiar, intuitive, and unchanged debugging flow combined with a minimal learning curve for the user. Moreover, maintainance and customization of the flow are reduced to a minimum.

Debugging at system level requires various kinds of high-level information that should be retrievable fast and easily. According to [DSG03], one main information category is of interest in the debugging context:

Run-time infrastructure information can be divided into three subcategories. (i) *Static simulation information* describes the structure of the architecture that means the number of modules, the number of processes and signals, the I/O interfaces and their connections, etc. (ii) *Dynamic simulation information* includes among other things the triggering conditions of processes, the process sensitivity lists, and the number and types of events in the simulation queue. (iii) *Debugging callbacks* (here, *high-level breakpoints*) allow to add callbacks from the simulation environment to break on certain events such as process activation, value changes on signals or the ongoing simulation time.

Studies at AMD indicated several debug patterns typically used in daily work. A pattern describes the steps (in gdb) to enquire needed debugging infomation at system level. Based upon characteristic debug patterns, high-level commands were implemented (table 1). At top level, commands are classified in examining and controlling types. In a distributed development flow, many designers are working on different components at the same design. In case of an error, it is essential to get a fast insight into external components and their interaction with your own ones. For this reason, examination commands retrieve either static or dynamic simulation information. Controlling commands provide high-level breakpoints to reach a point of failure at system level very quickly. They stop program execution at certain conditions, such as the next activation of a specific process or all processes which are sensitive to a given SystemC event.

Examining commands	
static simulation information	
lss	list all signals in given hierarchy
lsm	list all modules in given hierarchy
lse	output all events instantiated in modules
lsio	list I/O interface in given hierarchy
lsb	list all bindings of specified channel
dynamic simulation information	
lpt	list all trigger events of all processes (w.r.t. a specific time stamp)
lst	output code line a process is currently pending
lpl	show all processes listening on given event
lsp	output all [c]thread and method processes
Controlling commands	
ebreak	break on next invocation of processes that are sensitive to specified SystemC event
rcbreak	break on next invocation of processes that are sensitive to rising edge of given clock
fcbreak	break on next invocation of processes that are sensitive to falling edge of given clock
pstep	break on next invocation of given process
dstep	break on processes which will be active in the next simulation delta cycle
tstep	break on processes which will be active in the next simulation time stamp

Table 1: High-level debugging commands

2.2 General architecture

Figure 1 illustrates the layered architecture of our high-level debugging environment. Due to the demand for a non-intrusive extension of gdb, all high-level debugging commands are implemented on top of it. Command sequences are encapsulated as a unit in a user-defined command composing the so called **macro instruction set** at the user layer. A macro instruction implements a desired debug functionality by using built-in gdb commands (e.g. examining the symbol table, or the stack), and a set of additionally provided **auxiliary functions** at the API layer. Auxiliary functions are C++ or script helpers that evaluate and process information supplied by the **debug data pool** representing the data layer. The pool obtains its content either from redirected output of gdb commands (temporary log files), data structures of SystemC kernel classes, or a debug database holding preprocessed information collected during initialization of the actual debug session.

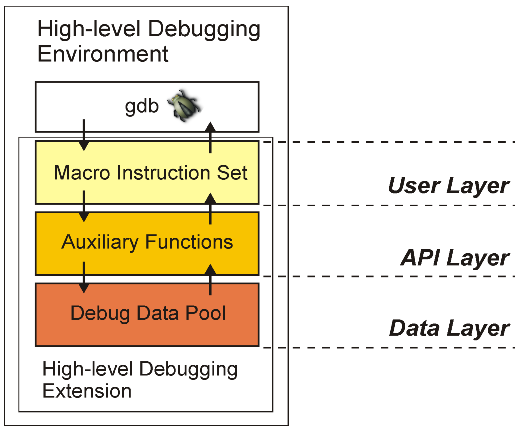

Figure 1: High-level debugging environment

2.3 Debug flow

Examining commands (table 1) mostly just process directly accessible information provided by the debug data pool. In contrast, a controlling command comprises a complex interaction between data provided by the pool and gdb. Here, execution starts usually with a redirection of a gdb command output (e.g. `backtrace`) into a temporary log file. The log content is evaluated by an affiliated auxiliary function. According to the specific debugging task, extracted log data trigger various actions:

- storing or updating data into the debug database,
- caching data in temporary data structures,
- retrieving enquired debug data from the database, or
- generating a temporary gdb command file.

A generated temporary gdb command file is sourced subsequently. Its execution releases either an instant action or it creates a (temporary) breakpoint which is triggered in the future. According to the specific task the loop of writing and evaluating log files, and performing various actions can run some further times. As a result, data are stored in the debug database, or enquired information is output at gdb console. In addition, oncoming debugging commands and data collection actions can be prepared by caching data, or setting (temporary) breakpoints.

Figure 2 sketches the exemplary execution of an imaginable debugging command. There, each participating component belongs to one of the three layers.

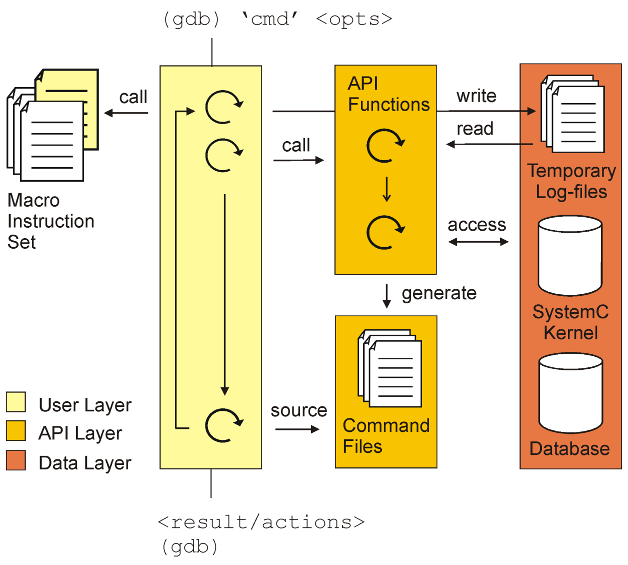

Figure 2: Exemplary debug flow

3 Implementation

3.1 User layer

The user layer acts as the interface to the high-level debugging capability. It comprises the macro instruction set which is summarized in several gdb script files. Furthermore, this layer contains gdb scripts to setup and to initialize the debugging environment.

Example. The lsb command (table 1) presents the common command implementation template at the user layer.

```
define lsb
if ($hd_elaborated)
   echo ---lsb: list all bound ports---\n
   call hd::list_bound_ports($arg0)
   echo -------------------------------\n
else
   echo not elaborated yet\n
end
end
document lsb
list all bindings for the specified channel
end
```

3.2 API layer

The API layer supports the implementation of high-level debugging commands at the user layer. It divides into an auxiliary function API and a database API.

The **auxiliary function API** comprises in addition to awk/shell scripts, particularly C++ functions which realise more sophisticated helper functionality. Scripts are normally used to straightforward process text files. Implementations of the same functions showed a significant performance yield of the C++ over the script based realisation.

The **database API** supplies functionality to store data into, and to retrieve data from the debug database. For each database type (section 3.3) a set of access functions is provided.

Example. A call of the lsb command invokes the C++ auxiliary function hd::list_bound_ports(const char) which retrieves the corresponding sc_interface instance using the SystemC method sc_simcontext::find_object(). Afterwards it calls hd::list_binding(sc_interface*). This database API function fetches the static binding information from the debug database and formats them accordingly for output (figure 3).*

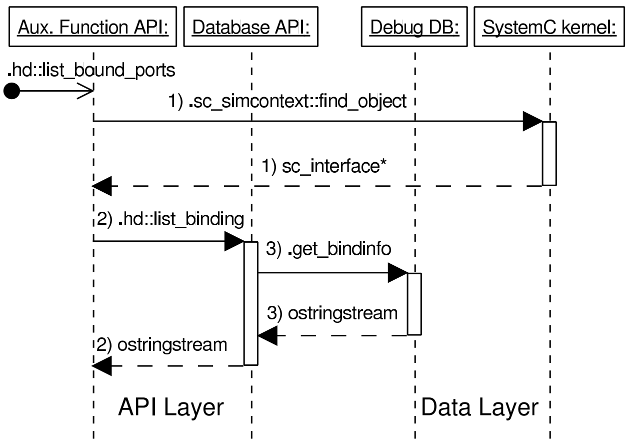

Figure 3: lsb command at the API layer

3.3 Data layer

Three data sources compose the data layer supplying either static or dynamic simulation information.

Temporary log files will be created by redirecting the output of gdb commands (e.g. `thread`, `backtrace`) providing *dynamic simulation information* only accessible at debugger side such as the assigned gdb thread ID of a SystemC thread process.

SystemC kernel The SystemC kernel provides some basic introspection capabilities useful for retrieving design and run-time information. Various global registration classes allow to query *static simulation information*, such as port, module, channel, or SystemC object registry. For instance, the object hierarchy can be easily browsed using the following loop:

```
sc_simcontext* c = sc_get_curr_simcontext();
sc_object* o = c->first_object();
while (o) {
   if(!strcmp(o->kind(),"sc_module")) {
      // module specific actions
   }
   else if(!strcmp(o->kind(),"sc_signal")) {
      // signal specific actions
   }
   ...
   o = c->next_object();
}
```

The simulation control, implemented by the kernel class `sc_simcontext`, supplies many valuable *dynamic simulation information* such as runnable processes at the next delta cycle, or the delta event queue.

Debug database During setup of a new debug session, *static simulation information* is logged and stored into the debug database using gdb (at least in the first implementation approach - section 3.4). Here, we utilize particularly the ability of a debugger to fetch private class data in the SystemC kernel which do not have public access methods. Required debug functionality (table 1) bases on four information classes: event, binding, method and thread process information. Each class is represented by its own datatype holding preprocessed (e.g. process entry function name), kernel-private (e.g. process handle private to `sc_process_table`), or special debug session data (e.g. gdb thread ID). Figure 4 sketches the UML class diagram of the debug database showing only the datatypes representing the information classes together with their attributes.

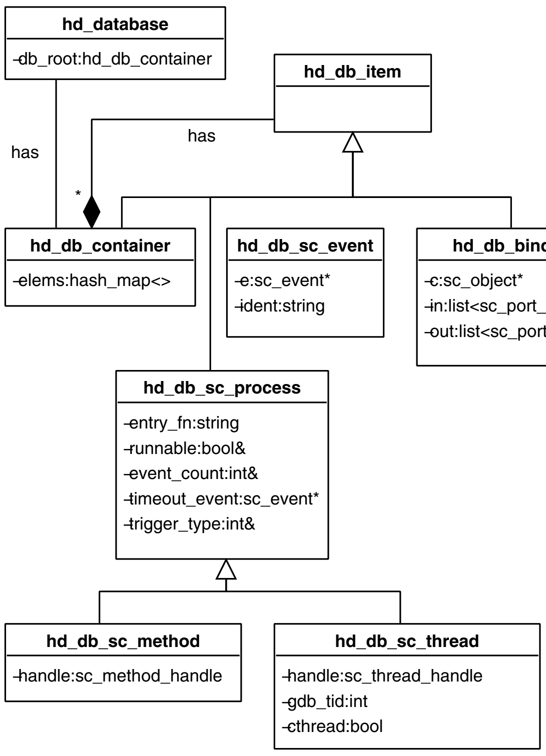

Figure 4: Debug database class hierarchy

Example. The following lsb call retrieves the binding information for a channel of an example application referenced by its hierarchical object name `i0_count_hier.count_sig`*. The database API queries for the proper hd_db_bindinfo instance using the corresponding* `sc_interface` *object, fetches, and formats its data.*

```
(gdb) lsb "i0_count_hier.count_sig"
---lsb: list all bound ports---
bindings of channel i0_count_hier.count_sig
Driver:
  i0_count_hier.i_counter.outp    <sc_out>
Drivee:
  i0_count_hier.i_signal2fifo.inp    <sc_in>
-------------------------------
```

3.4 Performance issues

Practical tests on real-world AMD applications revealed a considerable performance problem using the pure non-intrusive implementation approach. Especially, the setup phase of the extended gdb takes an unacceptably long time (table 2). Investigations indicated particularly the assemblage of the high-level debugging information as the bottleneck. Here, hidden breakpoints in the SystemC kernel registered and triggered actions to handle the instantiation of processes and events. So, we developed a second approach to accelerate the data assemblage phase. The idea is to reduce the number of breakpoints while moving their functionality into the kernel methods where the breakpoints were formerly set. Normally, one has to patch these methods to create callbacks forwarding required information into the high-level debugging environment. To remain kernel patch-free, we use library interposition and preload a shared library which overwrites appropriate SystemC kernel methods. There, the original implementation is extended by a callback into the debugging environment. A setting of LD_PRELOAD instructs the dynamic linker to use this library before any other when it searches for shared libraries (figure 5). Preloading works only for non-inlined class methods. Hence, minor transparent changes to SystemC kernel source code were necessary, i.e. moving inlined constructors of classes `sc_signal`, and `sc_event` from header to implementation files. Time measurements (table 2) document the efficiency of the new approach.

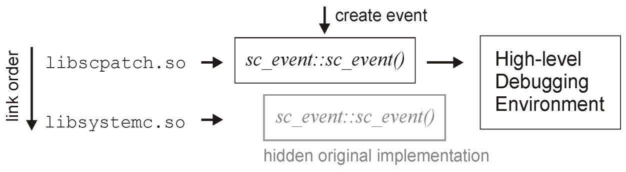

Figure 5: Preloading a SystemC kernel method

<table>
<tr><th rowspan="2">Test</th><th rowspan="2">gdb mem-usage</th><th colspan="3">Setup time</th></tr>
<tr><th><i>w/o</i></th><th><i>non-intrusive</i></th><th><i>pre-loaded</i></th></tr>
<tr><td>A[1]</td><td>45 MB</td><td>&lt; 1 s</td><td>2.25 min</td><td>1.2 s</td></tr>
<tr><td>B[2]</td><td>202 MB</td><td>~10 s</td><td>26 min</td><td>25 s</td></tr>
<tr><td>C[3]</td><td>208 MB</td><td>~10 s</td><td>&gt; 40 min</td><td>31 s</td></tr>
</table>

Table 2: High-level debugging environment setup

1. **Test system**: AMD Opteron™ 248 processor @2200 MHz, 3GB real memory, **Test application**: multiple instances of a simple producer/consumer application, *#threads: 204, #methods: 1010, #sc_events: 3638*
2. **Test system**: AMD Athlon™ XP processor @1800 MHz, 3 GB real memory, **Test application**: bus interface controller for various protocol implementations, #threads: 29, #methods: 56, #sc_events: 306
3. **Test system**: see 2., **Test application**: serial interface, #threads: 31, #methods: 136, #sc_events: 701

4 Practical application

4.1 Debug problem

As a short example we try to investigate why an interface bus of our design under test (DUT) has a value contention during simulation. It seems that there are two processes concurrently driving data onto the bus.

We know of the first process in our SystemC environment driving an initialization value after reset, namely **tb.bif.fsm._fsm_reset()**. This thread is sensitive to the negative edge of the reset signal **fsm_rst_l**, being a low active input to our DUT.

4.2 Conventional debug procedure

To find the second yet unknown process colliding with ours, we would set a breakpoint into the unique driver function that is invoked by every module that wants to stimulate the interface bus.

On any stop at this breakpoint we then had to trace back the invoking module, e.g. with the **up** command in gdb. This can turn out to be a time consuming task, potentially ending in different modules not involved in this specific issue. Since we know the signal event on which the problem occurs, a tracing of the signals SystemC event **fsm_rst_l.m_negedge_event** would help a lot.

4.3 High-level debug procedure

So we restart gdb with the high-level debugging environment to use the provided commands for event tracing (table 1). First, we set a breakpoint onto the observed signals negative edge event:

```
(gdb) ebreak "tb.bif.fsm_rst_l.m_negedge_event"
*** scheduled break on event
*** type <continue> to set breakpoint(s)
(gdb) continue
```

which, on the breakpoint stop, gives us two thread processes sensitive to it:

```
*** event 'tb.bif.fsm_rst_l.m_negedge_event'
triggered ...
   breakpoint at thread 21
   breakpoint at thread 16
(gdb)
```

Knowing that thread 21 is the FSM reset process, we look into the source code thread 16 is pending with **lst**:

```
(gdb) lst 16
--lst: list active source of [c]thread/method---
process tb.bif.if_bfm._update is currently
 at /home/hld/project/tb/src/if_bfm.cpp:128
in if_bfm::_update
126      get_bus_val(bus_val);
127      if_bus->write(bus_val && 0xff);
128      wait();
129   }
```

In line 127 we find a concurrent writing of a wrong value onto the bus. With the **lpt** command we review the threads sensitivity list:

```
(gdb) lpt
thread process sensitivity list
------------------------------
tb.bif.if_bfm._update
   <dynamic> tb.bif.ch_m._update.m_value_changed
   <static> tb.bif.fsm_rst_l.m_negedge_event
   <static> tb.bif.fsm_tx_w.m_posedge_event
```

We see that the sensitivity falsely includes also the reset signal, which is not desired and turns out to be an environment bug. Compared to section 4.2 we needed far less debug steps and straighter tracked down the issue.

5 Conclusion and future work

In this paper, we presented an environment to debug SystemC applications at high level working with signals, ports, processes, and events. High-level debugging commands realize debug patterns typically used at AMD. The special feature of our approach is its non-intrusive implementation that means it avoids real patches of the SystemC kernel or gdb sources. We apply library interposition to preload a shared library that allows to smoothly gather required debugging information provided by the SystemC kernel. Practical experiences in an industrial design flow show promising results with only a marginal increase of debug setup time.

Future work will improve the overall performance, and implement additional commands. Also, it would be necessary to increase the abstraction level of the commands in order to further simplify debugging at SystemC level. Furthermore, establishing a methodology or cookbook to apply high-level debugging commands could help the designer finding bugs more quickly.

References

[ARM06] ARM Ltd. MaxSim Developer home. *www.arm.com*. 2006.

[BWH03] F. Balarin, Y. Watanabe, H. Hsieh, L. Lavagno, C. Passerone, A. Sangiovanni Vincentelli. *Metropolis: an integrated electronic system design environment*. IEEE Computer, Vol. 36, No. 4, April 2003.

[Che03] K.-C. Chen. *Assertion-Based Verification For SoC Designs*. Proceedings of 5th International Conference on ASIC, Vol.1, Oct 2003.

[CRA01] L. Charest, M. Reid, E.M. Aboulhamid, G. Bois. *A Methodology for Interfacing Open Source SystemC with a Third Party Software*. DATE 01, March 2001.

[DSG03] F. Doucet, S. Shukla, R. Gupta. *Introspection in System-Level Language Frameworks: Meta-level vs. Integrated*, DATE 03, March 2003.

[GDB06] GDB home. *www.gnu.org/software/gdb*, 2006.

[GDL03] D. Große, R. Drechsler, L. Linhard, G. Angst. *Efficient Automatic Visualization of SystemC Designs*. FDL'03, Sept 2003.

[GHJ99] E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design Pattern - Elements of Reusable Object-Oriented Software*. Addison Wesley Professional Computing Series, 1999.

[Gra06] GRACE++ project home. *www.iss.rwth-aachen.de/Projekte/grace/visualization.html*. 2006.

[GZD00] D.D. Gajski, J. Zhu, R. Domer, A. Gerstlauer, S. Zhao. *SpecC: Specification Language and Methodology*. Kluwer Academic, 2000.

[SCI06] Open SystemC Initiative home. *www.systemc.org*. 2006.

[SVH06] CoWare ConvergenSC System Verifier home. *www.coware.com*. 2006.

[Syn06] Synopsys System Studio home. *www.synopsys.com*. 2006.