AIBA: an Automated Intra-Cycle Behavioral Analysis for SystemC-based Design Exploration

Mehran Goli[1] Jannis Stoppe[2] Rolf Drechsler[3]
[123]Institute of Computer Science, University of Bremen, 28359 Bremen, Germany
[23]Cyber-Physical Systems, DFKI GmbH, 28359 Bremen, Germany
{[1]mehran [2]jstoppe [3]drechsle}@informatik.uni-bremen.de

***Abstract*—In order to overcome the ever increasing complexity of digital circuits, system design at the *Electronic System Level* (ESL) has become an area of active research.**

SystemC provides designers with a readily-available ESL framework, allowing them to design mixed hardware/software systems using a standardized C++ library. The analysis of the resulting designs is crucial to e.g. apply additional validation steps or assist designers during the development process. Existing approaches focus on the extraction of static information, providing designers with models that describe the structure of their system but not its behavior. In this paper, we introduce the *Automated Intra-cycle Behavioral Analysis* tool, *AIBA*. *AIBA* utilizes the GNU debugger to execute a two-step analysis that retrieves behavioral and architectural information of ESL designs.

The proposed method is completely non-intrusive, allowing both SystemC designs and the standard tool flow to be used without any modification. Case studies confirm the benefits of the approach.

I. Introduction

One approach to handle the increasing complexity of circuits and systems is the utilization of more abstract description means. The Electronic System Level (ESL) [10] has been introduced as an intermediate layer between formal and informal specification languages and the synthesizeable Register-Transfer Level (RTL) [18] descriptions. SystemC [2] has emerged as a de-facto standard [13] to specify designs at the ESL.

Understanding the respective components of the design as well as their relation to each other is a crucial and – due to the SystemC's C++ foundation – non-trivial task. Moreover, knowing the dependencies within hardware systems has always been a key issue in pre and post design process in terms of debugging, system exploration, visualization, static verification and synthesis tools.

According to [11], SystemC design extraction methods should satisfy the following criteria:

- assume as few a priori restrictions as possible,
- retrieve precise information on all parts of the model,
- maximize code reuse to avoid creating new C++ dialects and to ensure that the approach is also applicable to future SystemC compilers.

Especially the second point leaves room for interpretation – the kind and amount of information to be retrieved is usually depending on the task at hand and thus varies with the use case being pursued. While current approaches are retrieving detailed static models, they do not extract any behavioral information.

Existing approaches utilize the infrastructures such as custom(ized) front-ends [4], [12], [3], compilers/parsers [7], [8], [11] or end-user modifications to the source code or manual intervention (e.g. using a software debugger [14]). All these approaches restrict the development options.

AIBA provides an applicable, non-intrusive approach in order to retrieve both, a static model and behavioral information of a given SystemC design. It uses a two-step approach that relies on the GNU debugger (GDB) [14]:

1) The static information of a SystemC model is retrieved from the compiled, executable file by analyzing its debug symbols.
2) The dynamic (i.e. behavioral) information is retrieved from the executable design at run time. By creating a GDB source command file based on the static information that instructs the debugger to retrieve any relevant values and executing the design under control of the debugger using these instructions, a detailed model of the given design is created and stored.

The retrieved data includes the design hierarchy and detailed run-time traces and is presented as a Value Change Dump (VCD) file. It thus can be browsed using established tools.

II. Related Work

Several methods for SystemC data extraction have been proposed.

Static methods extract the information of a design from the source code, i.e. they do not rely on the execution of the given design's elaboration phase. Approaches can be divided into parsers [6], [8] or existing C++ front-ends [4], [12], [3]. The results are restricted to the static information of the model that in the best case can be presented in an Abstract Syntax Tree (AST), thus disregarding the behavioral properties of the design.

In addition to the static data extraction, hybrid methods analyze the dynamic behavior of a given SystemC program.

Pinapa [11] uses two phases to extract the information. First by parsing the SystemC model using GCC to retrieve the AST. Secondly, it executes the elaboration phase of the model to extract the architectural information. Then, the extracted information is merged to generate the final result. It has limitations such as the inability to generate useful output when the SystemC model includes some constructs such as pointers to SystemC objects or complex array index expressions. Moreover, Pinapa cannot extract the order of function calls and process activation during the execution of the model.

In [7] a hybrid technique is introduced which uses a PCCTS based parser to retrieve the static information and a code generator to extract run-time information. The method is split into four steps which converting the SystemC model into an AST, creating the instrumented version of the original model, executing elaboration phase of the model and finally recording the state of all variables of the model. Using the PCCTS based parser limits the available SystemC constructs though.

PinaVM [9] uses the LLVM compiler to translate the SystemC source code into LLVM bit-code. To extract the dynamic information, PinaVM first specifies the parts of the source code which contain the parameter of interest (e.g. dynamic information of a module's ports) and then constructs new functions to retrieve it which replace the according original behavior during the compilation of the program. This method is limited to models in which the structure can be determined statically. Additionally, using the LLVM project as a foundation limits PinaVM to setups that are built using LLVM.

SHaBE [5] uses GDB to retrieve the static information and a GCC plugin to retrieve the dynamic information. It retrieves an AST and links the dynamic information with the extracted module hierarchy. Concerning to the dynamic information, the approach has some limitations concerning the extraction of hierarchical information such as SystemC primitive channels (e.g. clock signal, fifo and semaphore), some SystemC module hierarchies and the information of a process sensitive to an event from channel.

Another recently published hybrid method [16] uses debug symbols to extract the static information and SystemC API calls to retrieve dynamic data during the execution of a SystemC model. The method does not capture behavioral data but only uses the SystemC API to dynamically build a static hierarchy of a SystemC model.

III. Motivation

Existing approaches have two major limitations in terms of dynamic data extraction and restricting the language. The first limitation is that most of them only retrieve the module hierarchy and restricted dynamic information (e.g. instance names of modules and a discrete state of variables). This is not necessarily sufficient to properly understand a given system. The second limitation is that most of them can only be applied to a restricted range of SystemC designs.

We are aiming at overcoming the given drawbacks of traditional SystemC data extraction and providing a method to retrieve detailed behavioral information from a given design. More precisely, we consider the question:

> *How can both, the structure* and *the behavior of a given SystemC design be retrieved without restricting the language means and/or modifying the existing infrastructures?*

There are several approaches that potentially enable the extraction of behavioral data. One of them, is Aspect-Oriented Programming (AOP) which is a paradigm that allows the designer to write refactoring rules that are applied before compiling a program (a process called weaving). AOP was already applied in several SystemC analysis projects [15], [17]. This approach comes with several pitfalls. First, debugging AOP setups is a complex task. Furthermore, AOP does not support the description of arbitrary points that define the code that should be altered, which is a deal breaker for the goal of arbitrary behavior tracing.

Another alternative, is Clang/LLVM. It is possible to extract static information from the abstract syntax tree generated by the Clang compiler and applying plugins to the compilation workflow in order to make the program extract its own dynamic information by injecting new functions into the program at compile time. Using the LLVM infrastructure limits the approach to setups that rely on LLVM, which not only excludes the proprietary Microsoft front-ends, but also projects that rely on GCC, which is a serious drawback to this approach.

By utilizing GDB as an underlying infrastructure, several shortcomings of other approaches can be solved. First, GDB provides a toolset to monitor the behavior of a design during the execution of a model down to single instructions. It is thus possible to access information per instruction during execution, addressing the issue of arbitrary behavior extraction. Secondly, GDB is compatible to both, GCC and Clang/LLVM and has no limitation by means of SystemC languages, which increases compatibility compared to other SystemC analysis approaches and the LLVM-plugin approach (which would be incompatible to non-LLVM compilers).

The approach presented in this paper on the other hand provides a novel kind of information to be extracted from SystemC designs. In addition to the static structures, it allows designers to extract the precise behavior of a given SystemC model. It provides designers with a new aspect of dynamic information extraction called *intra-cycle behavior*: by shifting the extraction methods to an underlying debugger, the behavior of a system within single points in simulation time can be retrieved from any given data type.

IV. Proposed Method

Fig. 1, represents the architectural view of AIBA. As stated in this figure, AIBA is based on two main phases,

1) The extraction of
   - static information and
   - the data required to generate a set of GDB instructions

   from a compiled SystemC design that includes the required debug symbols and
2) the retrieval of the dynamic information of the model by executing it via GDB, using the previously generated GDB instructions.

Finally, the retrieved information is stored as a VCD file.

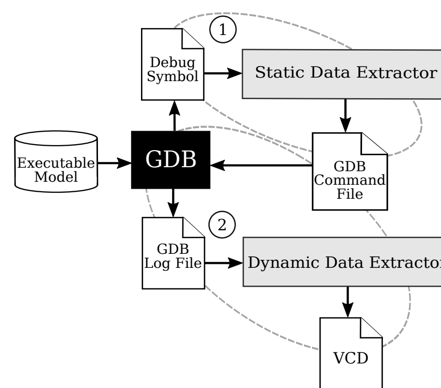

Fig. 1. The architecture of AIBA.

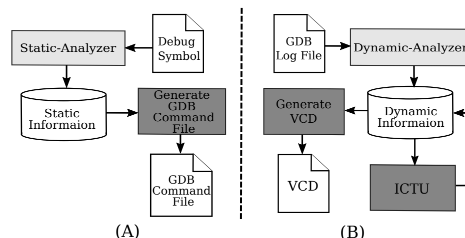

Fig. 2. The architecture of A) Static Data Extractor and B) Dynamic Data Extractor.

A. Extracting Static Information

The static information refers to structural data that is described in the design's source code. This data consists of class and function names, variable information (e.g. module ports, local variables of functions), class hierarchy information, data types, etc.

The *Static Data Extractor* module retrieves the static information from the GDB debug symbols as shown in Fig. 1 in phase 1. In addition to being helpful information by itself, the gathered static data is used as the foundation to retrieve the dynamic run-time information in the next step. Based on the static data, the *Static Data Extractor* module automatically generates a *GDB Command File (GCF)* which is required to extract the dynamic information in the second phase.

Based on the information provided by accessing the debug symbols, the *Static-Analyzer* module extracts the static information of the model. Fig. 2.A illustrates the architecture of the *Static Data Extractor* module, reading the debug symbols, translating them into a more manageable data format and finally translating them into a source file to utilize the information about the design's static structures to retrieve the dynamic information as well.

B. Extracting Dynamic Information

Behavior analysis is usually done using VCD files that are generated using the standard SystemC API. This approach works well for SystemC signals (which represent hardware signals) but lacks precision for base type variables that may change several times during a single SystemC-δ-cycle (the smallest amount of time that may pass concerning the simulation kernel) and fails for user-defined datatypes that are not supported at all unless the designers alter their code. In order to overcome this drawback, the program is executed via GDB, which is usually done to manually debug a design by setting breakpoints and stepping through the program. In order to automate this approach, the previously generated GDB command source file contains instructions for the debugger to automatically halt the execution of a program, store any changed values and resume the execution afterwards.

As this approach does not rely on calls from the SystemC kernel, but instead uses GDB as an execution environment, the smallest

timestep to differentiate assignments are C++ statements instead of SystemC δ cycles. This results in a (potentially) higher precision, tracking value changes in the order they occur in, while still setting them in context of the current simulation time. This behavior of values within a single δ cycle shall be called the *intra-cycle behavior*, and its extraction is the focus of this chapter.

1) Generating the GDB Log File: The SystemC model is executed via GDB, utilizing the generated *GCF*. Based on the commands within the *GCF*, GDB sets breakpoints for each function of the given modules and the program continues to hit the first breakpoint. The commands of the breakpoint which have been hit is executed and calls the corresponding GDB function for this breakpoint.

As is illustrated in Fig. 2.A the extracted information is stored in an internal data structure (the *Static Information*). Each module is described in a hierarchical format based on its member functions and attributes. In order to retrieve behavioral information of the model, a combination of the static information contained in the *Static Information*, with the scheme and basic commands of GDB, are utilized by the *Generate GDB Command File* module to create a *GCF*. This module defines a breakpoint for each function of a module based on its name and scope and a command for each breakpoint in the *GCF*. These breakpoints are scripted to automatically execute a GDB function call which is defined separately for each module function in the *GCF*. These functions contain instructions for the debugger to retrieve the dynamic information of the SystemC model during its execution.

More precisely, for each breakpoint, the GCF contains instructions for GDB to retrieve:

- the simulation time,
- the value of all variables of the current module and all values of local variables of the current function,
- the name of each port,
- the instance name of current module,
- clock information (if available) and
- binding information of each port.

Algorithm 1: ICTU update process

foreach *time unit in Dynamic Information* **do**
  vc[time unit] = sum (number of value changes of all variables);
end
max = maximum (vc);
$\mu_t = (1/max) *$ simulation time scale;
foreach *time unit* t *in Dynamic Information* **do**
  foreach *variable* **do**
    if *the number of variable's assignments > 1* **then**
      foreach *value assignment* **do**
        $t_{new} = t + \mu_t$;
        $store\ (t_{new}$, value assignment);
        $t = t_{new}$;
      end
    end
  end
end
update ($Dynamic\ Information$);

2) Generating VCD: As illustrated in Fig. 2.B, the dynamic information of the model is extracted by the *Dynamic-analyzer* module. The simulation time stamp is captured as well as the value of variables when an instruction is executed. All retrieved dynamic information is stored in an internal data structure called the *Dynamic Information* as seen in Fig. 2.B. In each time unit in the *Dynamic Information*, a list of variables that was extracted at this time is stored. The *Dynamic-analyzer* module binds all ports of a module with the corresponding signals that have a same interface.

In the next step, the information of the *Dynamic Information* is processed by the *Intra-Cycle Time Unit (ICTU)* module to differentiate values of variables within a single point in simulation time. To present all events on a single timeline, the assignments of each single point in simulation time are sorted by their execution order o and the value $o \cdot \mu_t$ is added to their original timestamps. The value μ_t is thus used to differentiate the particular assignments. It should be much smaller than the smallest step in simulation time in order to have all assignments being displayed before the next "large" simulation timestep. μ_t is therefore related to the maximum sum of the number

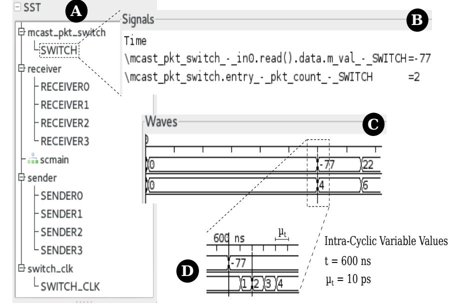

Fig. 3. A part of generated VCD file for the pkt_switch example.

of value changes of variables in a time unit among all time units as illustrated in Algorithm 1 and is calculated automatically.

Finally, the *Generate VCD* module gets the modified *Dynamic Information* from the previous step and generate a VCD file. It thus provides a time-ordered sequence of value changes based on the extracted simulation time unit.

V. Evaluation

Both the *Static Data Extractor* module and the *Dynamic Data Extractor* module have been implemented in Python. The proposed method has been used GCC 4.5 and GDB 7.1. All execution times have been measured on a PC equipped with 8 GB RAM and the Intel core i7-2760QM CPU running at 2.4 GHz.

A. Case Studies

In order to illustrate the advantages of AIBA in visualizing a wide range of ESL designs, a variety of SystemC models have been analyzed. The SystemC models that are presented in Table I are taken from the standard examples which provided by OSCI and University of Edinburgh.

The `pkt_switch` example realizes a system for distributing data packages and is assumed to be instantiated with four sender and four receiver instances, all of them connected to a central switch.

Fig. 3 illustrates a part of the generated VCD file of the `pkt_switch` system. The static and dynamic information of the mentioned design has been retrieved and is shown in four parts in this figure.

(A) shows a basic hierarchy of the `pkt_switch` system in form of the *Signal Search Tree (SST)*. It includes the name of modules and their instances as well as the global functions (e.g. *scmain* function in this example).
(B) illustrates information of variables within the design. Each variable is identified based on its hierarchical structure which consists of the name of the root module, the name of function (for local variables), the name of variable and the instance name of the root module.
(C) shows the current value of each variable in the shape of a waveform with respect to the simulation time.
(D) demonstrates the presentation of intra-cycle behavior for the `pkt_switch` system. In this example, in time unit $t = 600$ns the local variable `pkt-count` is assigned four different values consecutively. It starts with its value being 0. By executing the next instructions it gets raised from 0 to 4 in four steps. To cover these temporary changes in a single time unit, the *Dynamic Information* is analyzed by ICTU module based on Algorithm 1. In this example, the maximum amount of assignments within each of the simulation timesteps is $max = 100$ and the simulation time scale is $ts = 1$ns. Based on this, the smallest step within a single time unit is $\mu_t = \frac{ts}{max} = 10$ps.

Table I summarizes the results obtained from the case studies. The first two columns show the SystemC models as well as the number of lines of code for each design respectively. The results of AIBA are presented in comparison with the SystemC trace file in terms of the

TABLE I
CASE STUDIES

SystemC Model	# Lines	SystemC Trace #Var	SystemC Trace #Time Units	SystemC Trace Exec(s)	AIBA #Var	AIBA #Time Units	AIBA Exec(s)	TP-GDB Exec(h)
1-bit Full Adder[1]	87	5	5	0.01	18	14	1.9	> 1
4-bit Shift Register[1]	135	3	29	0.02	12	54	2.2	> 1
FIR Filter[2]	233	6	24	0.02	23	504	4.1	> 3
Packet Switch[2]	1020	55	1723	0.05	332	5606	19.2	> 10
RISC CPU[2]	1960	89	137	0.03	254	551	12.1	> 10

[1] Provided by University of Edinburgh [1]. [2] Provided by OSCI.

number of retrieved variables #*Var*, number of extracted time units #*Time Unit* and execution time #*Exec*.

B. Integration and Discussion

As illustrated in table I, the amount of both, extracted variables and time units for all case studies of AIBA are much higher than those retrieved via the SystemC trace file method. The parameter #*Time Unit* is generated based on value changes of variables which have been retrieved during the execution of an executable model. Therefore, both of these parameters can show the accuracy of our approach to extract the detailed behavior of a SystemC design.

Although the SystemC comes with a trace API that enables designers to monitor the behavior of a SystemC model via VCD logs, it has some severe limitations. The information extraction of some primitive channels and module ports (e.g. `sc_in`, `sc_out`, `sc_inout` are not supported off-the-shelf). In addition, using the SystemC trace function to analyze the behavior of a given SystemC model is an intrusive solution: the original source code needs to be modified by the designer to include all values that need to be traced, which, for a complex design with lots of variables, may be a time-consuming task.

Table I also compares the execution time of AIBA to the SystemC trace method and the trace point functionality by GDB. GDB's trace point feature can be used to retrieve the detailed behavior of a model but even for a simple SystemC model, the required execution time lies in an order of hours. The method also needs to be set up manually which makes the solution inapplicable even for a simple model.

The proposed approach thus represents a trade-off between the precise information extraction and the execution time. As demonstrated in Table I, AIBA provides the intra-cycle behavior of a given SystemC model in an reasonable run-time.

The only precondition for the application of the suggested approach is that the executable SystemC model contains debug information. The original source code and SystemC library do not need to be modified in any way which makes it a non-intrusive approach and can be applied to future versions of SystemC. Overall, this makes our approach applicable for a wide range of applications of SystemC models in ESL design.

VI. CONCLUSION

In this paper, we have introduced an automated intra-cycle behavioral analysis, called AIBA to extract both static information and the behavior of a given SystemC model. The experimental results show that AIBA provides a more detailed and less intrusive solution than the SystemC trace file API which is the only existing method to generate a VCD form of a SystemC model. This can be effectively utilized for debugging, system exploration, visualization, static verification or synthesis tools. Furthermore, compared to the trace point feature by GDB (as a solution to analyze the behavior of a SystemC model), AIBA automatically retrieves the precise behavior of a SystemC model in reasonable time.

In summery, AIBA can automatically give an access to the intra-cycle behavior of a given SystemC model, while other existing approaches do not support this feature. The presented method can be applied without any modification to the source code of the model or the library. This feature makes AIBA non-intrusive and applicable as long as the model is built including GDB-compatible debug symbols.

VII. ACKNOWLEDGMENTS

The research reported here was supported by the German Federal Ministry of Education and Research (BMBF) under grants 01IW13001 (SPECifIC) and 01IW16001 (SELFIE), German Research Foundation (DFG) within the Reinhart Koselleck project DR 287/23-1, and University of Bremen's graduate school SyDe, funded by the German Excellence Initiative.

REFERENCES

[1] http://www.inf.ed.ac.uk/teaching/courses/inf2c-cs/13-14/labs/lab2.html.
[2] IEEE Standard SystemC Language Reference Manual. *IEEE Std 1666-2005*, pages 1–423, 2006.
[3] D. Berner, J. piere Talpin, H. Patel, D. A. Mathaikuty, and E. Shukla. SystemCXML: An extensible SystemC front end using XML. In *Forum on specification and Design Languages (FDL)*, pages 405–409, 2005.
[4] N. Blanc, D. Kroening, and N. Sharygina. Scoot: A tool for the analysis of SystemC models. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 467–470. Springer, 2008.
[5] H. Broeders and R. Van Leuken. Extracting behavior and dynamically generated hierarchy from SystemC models. In *Design Automation Conference (DAC)*, pages 357–362. ACM, 2011.
[6] G. Fey, D. Große, T. Cassens, C. Genz, T. Warode, and R. Drechsler. Parsyc: An efficient SystemC parser. In *Workshop on Synthesis And System Integration of Mixed Information technologies (SASIMI)*, pages 148–154, 2004.
[7] C. Genz and R. Drechsler. Overcoming limitations of the SystemC data introspection. In *Design, Automation and Test in Europe Conference Exhibition (DATE)*, pages 590–593, April 2009.
[8] D. Große, R. Drechsler, L. Linhard, and G. Angst. Efficient automatic visualization of SystemC designs. In *Forum on specification and Design Languages (FDL)*, pages 646–658. Citeseer, 2003.
[9] K. Marquet and M. Moy. Pinavm: a SystemC front-end based on an executable intermediate representation. In *Embedded software (EMSOFT)*, pages 79–88. ACM, 2010.
[10] G. Martin, B. Bailey, and A. Piziali. *ESL Design and Verification: A Prescription for Electronic System Level Methodology*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007.
[11] M. Moy, F. Maraninchi, and L. Maillet-Contoz. Pinapa: An extraction tool for SystemC descriptions of systems-on-a-chip. In *Embedded Software (EMSOFT)*, pages 317–324, New York, NY, USA, 2005. ACM.
[12] T. Schubert and W. Nebel. The quiny SystemCTM front end: Self-synthesising designs. In *Selected Contributions from Forum on specification and Design Languages (FDL)*, pages 93–109. Springer, 2007.
[13] C. Schulz-Key, M. Winterholer, T. Schweizer, T. Kuhn, and W. Rosentiel. Object-oriented modeling and synthesis of SystemC specifications. In *Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 238–243, Jan 2004.
[14] R. Stallman and C. Support. *Debugging with GDB: The GNU Source-level Debugger*. Free Software Foundation, 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301, USA, ninth edition, 2010.
[15] J. Stoppe and R. Drechsler. Analyzing SystemC designs: SystemC analysis approaches for varying applications. *Sensors*, 15(5):10399, 2015.
[16] J. Stoppe, R. Wille, and R. Drechsler. Data extraction from SystemC designs using debug symbols and the SystemC API. In *VLSI (ISVLSI), 2013 IEEE Computer Society Annual Symposium on*, pages 26–31. IEEE, 2013.
[17] J. Stoppe, R. Wille, and R. Drechsler. Automated feature localization for dynamically generated SystemC designs. In *Design, Automation and Test in Europe Conference and Exhibition (DATE)*, pages 277–280. IEEE, 2015.
[18] F. Vahid. *Digital Design with RTL Design, Verilog and VHDL*. Wiley Publishing, 2nd edition, 2010.