Journal of Computer Science 6 (12): 1505-1510, 2010
ISSN 1549-3636

A Frame Study for Post-Processing Analysis on System Behavior: A Case Study of Deadline Miss Detection

Junghee Lee and Jongman Kim
School of Electrical and Computer Engineering, Georgia Institute of Technology,
777 Atlantic Drive NW, Atlanta, GA 30332, USA

Abstract: Problem statement: A lot of data can be obtained by system simulation using transaction level models without affecting the performance of the system. Due to huge amount of the raw data, we often need to post-process them to extract valuable information. Profiling capabilities of commercial tools provide predefined functionalities and don't allow users to add or modify for their own purpose. **Approach:** This study proposed a general frame study for the automation of post-processing simulation results using Boolean representation. The proposed frame study consists of Boolean expresser, manipulator and analyzer. **Results:** The frame study was illustrated with a case study of deadline miss detection. **Conclusion:** The frame study was practical as it provides flexibility, generality and ease of use.

Key words: Simulation result, post-processing, Boolean expresser, signal deadline, assertion checker, manipulator, rising edge, raw data, temporal logic, frame study, utilization monitors

INTRODUCTION

System simulation using Transaction Level Models (TLMs) (Ghenassia, 2005) is gaining more popularity for analyzing and optimizing a system. A lot of raw data can be obtained without affecting the performance of the system. However, often we need to post-process the raw data to extract valuable information from them due to their huge amount.

To automate post-processing, we need to represent all the system events by a unified form. However, characteristics of events are not uniform. For example, hardware events like bus transactions can be easily represented by Boolean but software behaviors are not. They may be converted into linear temporal logic (Pnueli, 1977) or computation tree logic (Clarke and Emerson, 1981). Such logical representations are good for formal verification but impose limitations on expressiveness.

Therefore, this study proposes a general frame study for post-processing that adopts Boolean representation. All the system events are represented as Boolean along with manipulators, which is the main contribution of this study. The Boolean representation itself traces only when and whether an event occurs. Its semantics is interpreted by manipulators. The Boolean representation is simple to use but expressive enough to represent the simulation results for analysis purpose.

Figure 1 shows the proposed frame study that has three components: Boolean expresser, manipulator and analyzer. First, the Boolean expresser converts the simulation results into the Boolean representation. The manipulator interprets the semantic of the Boolean representation and manipulates it to provide the analyzer with inputs. Finally, the analyzer generates analysis results. The Boolean expresser and the analyzer can be reused as libraries and the manipulator handles the application specific features.

Typical situation of using our frame study is as follows. Simulation models of hardware and software have been built. Simulations are conducted many times varying parameters of hardware and software or modifying a part of software. To analyze results of each run of simulation, our frame study is used.

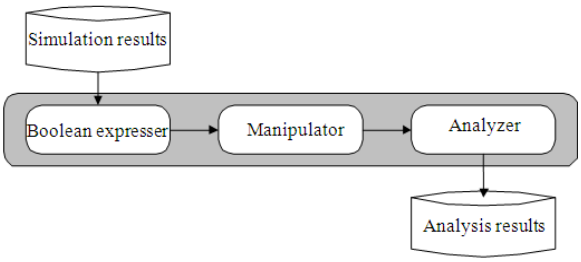

Fig. 1: Proposed framework

Corresponding Author: Junghee Lee, School of Electrical and Computer Engineering, Georgia Institute of Technology, 777 Atlantic Drive NW, Atlanta, GA 30332, USA

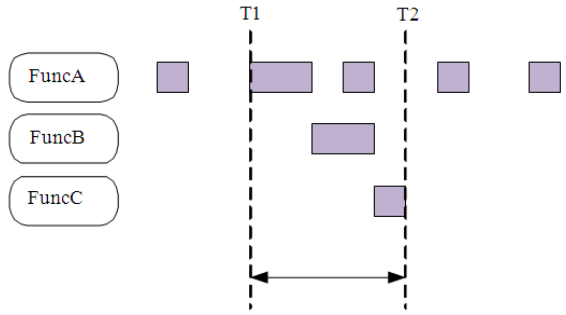

Fig. 2: Motivating example

Motivation and related studies: Figure 2 shows a motivating example, a Gannt chart of FuncA, FuncB and FuncC. The gray box indicates that the function is being executed. FuncA is executed periodically and triggers FuncB on a certain condition. FuncC runs right after FuncB finishes. Suppose that we are to measure the execution time between T1 and T2. T1 is the start time of FuncA just before FuncB starts and T2 is the end time of FuncC.

To automate measuring such a point, a specification method is necessary. Conventional software profilers provide execution time of each function or statement (Stewart and Arora, 2003). Since functions and statements have regular structure, they don't need to be specified explicitly. However, in this example, we need a specification method because the point to be measured is application specific.

Instrumentation can be used for measuring application specific points. The instrumentation code is inserted into the target program to be measured either statically at compile-time (Stewart and Arora, 2003) or dynamically at run-time (Corliss *et al*., 2005). Measurement can be done by post-processing the raw data or by the executable assertion (Pinter and Majzik, 2005; Drusinsky *et al*., 2005) implemented in the instrumentation code.

The drawbacks of the instrumentation are that it can be applied only for the software and that it may cause distortions because it should be inserted into and executed with the target program. Even small overheads of the instrumentation code may result in a distortion of the operating sequences of the system. More instrumentation codes can facilitate debugging and optimizing, but they result in additional distortions, which inhibit the insertion of as many instrumentation codes as required. Thus, there have been studies to reduce the overhead of the instrumentation (Metz *et al*., 2003). Hardware-assisted measurement does not cause distortion, but its precision and scope are limited (Stewart and Arora, 2003).

Therefore, system simulation using TLMs (Ghenassia, 2005) is getting more attention. The use of simulation results makes it possible to measure the performance of a system non-intrusively without any limitations on the precision and scope. We can obtain a lot of raw data by the simulation. Because of huge amount of simulation results, often we need post-processing them to extract valuable information.

There are commercial simulation tools (SoC Designer, http://www.arm.com; Innovator. http://www.synopsys.com; System Generator. http://www.arm.com; CoMET. http://www.vastsystems.com) providing analysis capability. They provide predefined individual analysis facilities. They don't provide a way to specify application specific measurement. Practitioners often use their own scripts for application specific measurement. In most cases, they are designed case by case. There is no general frame study to our best knowledge.

MATERIALS AND METHODS

This study describes two examples of the Boolean representation: function call trace of software and bus transactions. The way to convert the call trace and bus transactions into the Boolean representation described in this section is not the only way. The frame study allows for the users to add their own Boolean expresser. Hardware traces of registers and signals don't need to be converted as they are already generated in Boolean form. However, sometimes we don't need every trace of registers and signals. The example of bus transactions is provided as an example of abstracting the simulation results.

The Boolean expresser converts simulation results into Boolean representation. Details of how to convert depend on the format of simulation results but the principle is same. The Boolean representation contains only when and whether events occur.

We found the Boolean expression is enough to express all the events for analysis purpose. There may be a simulation result that is not intuitively converted to the Boolean representation. A typical example is software trace. It is usually represented as a call trace. We often don't need all the information contained in the call trace. For an example of Fig. 2, what we need is when and which function is being executed, not which function calls which function. For that purpose, we may convert the call trace into the Boolean representation by representing which function is being executed as a signal whose value is true. More precisely, start of function is represented as a rising edge and end of

function is represented as a falling edge. In the same manner, other events can also be converted into the Boolean representation.

An example of converting the function call trace to the Boolean representation is as follows. In order to convert the function call trace to a Boolean representation, a set of call traces C and a set of function names N should be provided from the simulation. Three kinds of information are contained in the i-th element c_i: the name of the function n_i, a flag f_i to indicate whether the function is called or returned and a timestamp t_i. The name n_i should be an element of N and the flag f_i should be either 'C' or 'R', where 'C' and 'R' indicate call and return, respectively. Then, the procedure for converting the function call trace to a Boolean representation is given as follows:

1. For the given set of call traces C = {c_0, c_1, c_2, …, c_n} and the given set of function names N
   where $c_i = (n_i, f_i, t_i)$, $n_i \in N$, $f_i \in$ {'C', 'R'}
2. Make a set of signals S
   For ∀n∈ N, add a signal with name n into S
3. For ∀c_i∈ C
   If f_i is 'C'
   Make a rising edge of the signal s named n_i at t_i where s∈ S
   If f_i is 'R'
   Make a falling edge of the signal s named n_i at t_i where s∈ S

Figure 3 shows an example of the conversion procedure. The software in this example includes FuncA, FuncB, FuncC and FuncD. FuncA calls FuncB and FuncC consecutively and FuncC calls FuncD. Then, the set of function names N would be {'FuncA', 'FuncB', 'FuncC', 'FuncD'}. Function call traces from the execution of the software are shown on the upper right-hand side of Fig. 3. Each line corresponds to c_i and each column corresponds to n_i, f_i and t_i, respectively. For example, the first three rows are interpreted as follows: FuncA is called at T1, FuncB is called at T2 and then at T3, the function call trace returns to FuncA as FuncB is completed. The function call trace is represented as Boolean values like the waveform shown in Fig. 3. The function call of FuncA at T1 is represented as a rising edge of the signal FuncA at T1, the function call of FuncB at T2 is a rising edge of the signal FuncB at T2 and a return to FuncA from FuncB at T3 is represented as a falling edge of the signal FuncB at T3.

Bus transactions can be converted in a similar way for the given set of traces B and the set of transaction names N. A transaction means an explicit data exchange between a master and a slave (Ghenassia, 2005). The converted form has a higher abstraction level than the TLM in that it does not trace the data being exchanged. It is sometimes sufficient to trace the kind of transactions that are taking place with timestamps. In this case, f_i is 'S' or 'E', where 'S' and 'E' indicate the start and end of the transaction, respectively. The procedure is defined as follows:

1. For the given set of bus traces B = {b_0, b_1, b_2, …, b_n} and the given set of transaction names N
   where $b_i = (n_i, f_i, t_i)$, $n_i \in N$, $f_i \in$ {'S', 'E'}
2. Make a set of signals S
   For ∀n∈ N, add a signal with name n into S
3. For ∀b_i∈ B
   If f_i is 'S'
   Make a rising edge of the signal s named n_i at t_i where s∈ S
   If f_i is 'E'
   Make a falling edge of the signal s named n_i at t_i where s∈ S

Figure 4 shows an example of converting bus transactions to Boolean values. The bus trace in this example abstracts bus activities as three transactions: Request, Read and Write. In a Request transaction, the master requests bus access to the arbiter, while during the Read and Write transactions, the master performs data transfers on the bus. An example of bus transaction traces is shown in Fig. 4. The bus transactions are interpreted and converted in a similar manner as the function call trace. The waveform in the lower part of Fig. 4 shows the converted values of the bus transactions.

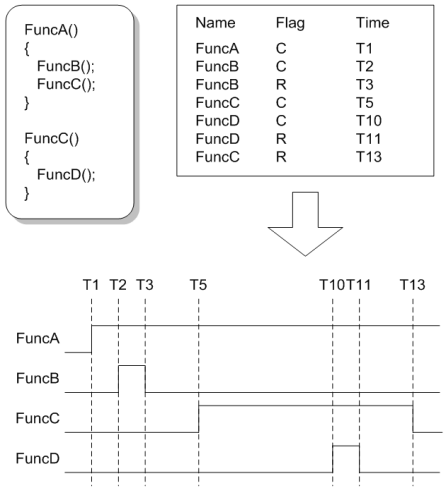

Fig. 3: An example of converting function call trace to Boolean values

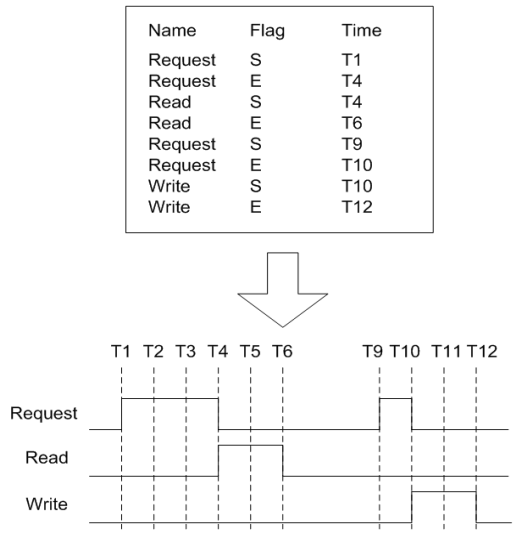

Fig. 4: An example of converting bus transactions to Boolean values

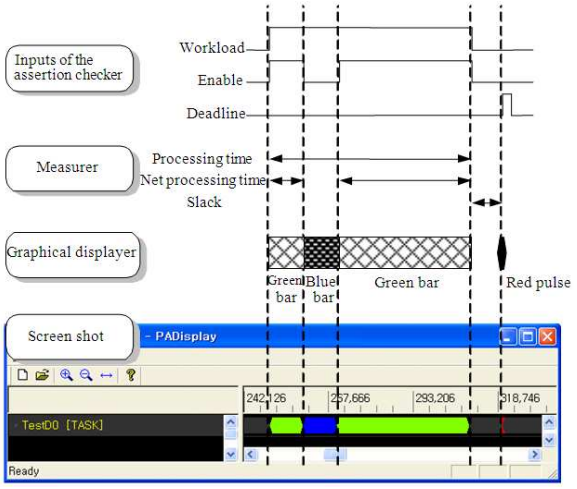

Fig. 5: Assertion checker

Two examples of the analyzer are described: the assertion checker and the utilization monitor. The analyzer described in this section can be reused for any application. The inputs to the analyzer should be provided by the user according to the application. The inputs to the analyzer are explained in the subsequent section.

The assertion checker is designed for detecting deadline misses. In this study, measurer and graphical displayer are also implemented. The assertion checker automatically detects whether a study load meets the deadline or not. A study load denotes a tuple of hardware and software tasks that have to be executed in order for a specific purpose. It should be noted that the formal definition of a tuple is a set that can contain an element more than once and the elements appear in a certain order. If a system comprises multiple study loads, every study load needs its own assertion checker so that its deadline can be verified.

Figure 5 illustrates the operations of an assertion checker. The inputs to the assertion checker are the signals Study load, Enable and Deadline. The signal study load indicates whether the study load to be checked is active or not. A rising edge indicates the start and a falling edge indicates the end of the study load. The signal Enable indicates whether the study load occupies all the necessary resources of the system or not. When its value is false, the study load is in the state of waiting for resources. For example, if a study load consisting of software tasks is preempted by an Interrupt Service Routine (ISR), the study load cannot be performed while the ISR is performed even though it is incomplete. This case would be represented as a false value of the signal Enable while the signal Study load is true. A rising edge of the signal Deadline indicates the deadline of the study load. The assertion checker determines a deadline miss by comparing a falling edge of the signal Study load and a rising edge of the signal Deadline. The signal Enable is not used for determining a deadline miss, but it is used for calculating the net processing time by the measurer described in the next paragraph.

The measurer calculates the statistics of the processing time, net processing time and slack of the study load. The processing time is the interval between a rising edge and a falling edge of the signal Study load. The net processing time is the processing time only when the signal Enable is true. Slack is the interval between a falling edge of the signal Study load and a rising edge of the signal Deadline. If a deadline miss occurs, the slack is calculated to be zero in this study.

The graphical displayer displays the status of the system graphically. A green bar indicates that a study load is actually performed, while a blue bar indicates that the study load waits for resources. If both the signals Study load and Enable are true, a green bar is displayed. A blue bar is displayed if the signal Enable is false and the signal Study load is true. If the signal Study load is false, nothing is displayed. A rising edge of the signal Deadline is displayed as a red pulse.

The utilization monitor is used for monitoring the variation of utilization with time. Generally, it is used for monitoring resources such as the CPU or bus. Figure 6 illustrates the operations of the utilization monitor.

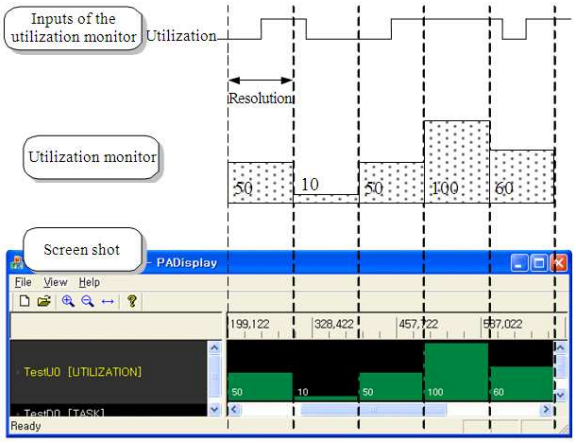

Fig. 6: Utilization monitor

The input to the utilization monitor is the signal Utilization. If the signal is true, it indicates that a resource is being used. The timeline is split into intervals whose duration is Resolution given by the user. In each interval, a bar is displayed; its height is the percentage of the summation of time when the signal Utilization is true within the interval over Resolution.

RESULTS

The results presents the case study of the deadline miss detection with the manipulator being illustrated. We consider a system that comprises the ARM926, Vectored Interrupt Controller (VIC), Personal Computer Memory Card International Association (PCMCIA), memory controller, modem and timer. The main function of the system is to transfer the data received from the modem to a host via PCMCIA. There are two study loads in the system. The modem study load receives data from the modem and moves it to the system memory. Rx interrupt is generated every one Millisecond (Ms) from the modem. The ISR (DataRxIsr), task Layer1and task Layer2 should be executed sequentially and completed before the next Rx interrupt is generated. In addition to the modem study load, the PCMCIA study load runs concurrently. When the host requests data, the Direct Memory Access (DMA) controller of the PCMCIA receives data from the system memory.

Figure 7 shows the manipulator for the modem study load. The manipulator can be considered as a behavioral hardware model. To provide the assertion checker with the input Study load, a sub module is designed. The rising edge of Study load should be identical to the rising edge of Data RxIsr and its falling edge is the falling edge of Layer2.

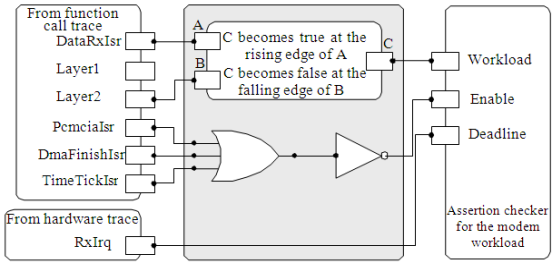

Fig. 7: Manipulator for the modem workload

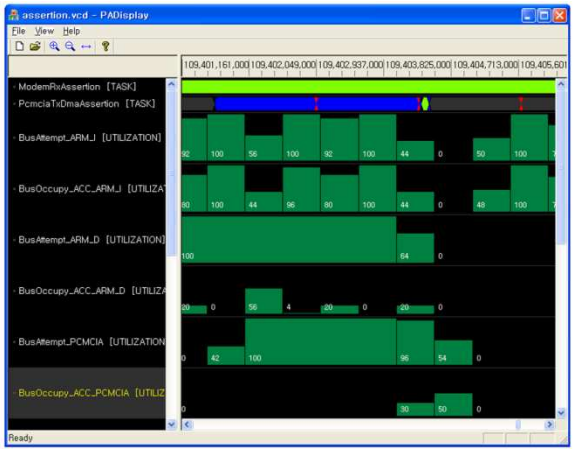

Fig. 8: Screenshot of the graphical displayer

Note that the sub module can be triggered at both rising and falling edges at the same time as it is like a behavioral model. The input Enable is made by logically OR-ing and inverting the ISR signals from the function call trace since the modem study load can be preempted by ISRs. The interrupt request signal from the modem to the VIC is directly used for the input Deadline. As for the PCMCIA study load, its assertion can be designed in a similar manner. Utilization monitors are added for the bus masters: the instruction bus of ARM926 (I-bus), the data bus of ARM926 (D-bus) and the DMA controller of the PCMCIA (DMA). If deadline misses are detected, utilization monitors may provide helpful information to investigate their causes.

Using the proposed technique, we could analyze simulation results of size 320 MB in a few min. It took 438 sec to analyze 24001001 cycles with the traces. The analysis speed was measured on an Intel 1.7 GHz Pentium M processor with 1.5 GB memory and Windows XP. Figure 8 shows a screenshot of the graphical displayer.

The inputs to the assertion checker are provided by logical operations of the Boolean representation. In

order to implement logical operations, a logic simulation engine was implemented using C++ in this study. The logical operations should be specified by the user using the Application Programming Interfaces (APIs) provided by the logic simulation engine in a manner similar to System C (Open SystemC Initiative, 2005). The Boolean expresser, assertion checker and utilization monitor were also implemented using the APIs and they can be used in a unified environment.

It should be noticed again that commercial simulation tools (SoC Designer, http://www.arm.com; Innovator. http://www.synopsys.com; System Generator. http://www.arm.com; CoMET. http://www.vastsystems.com.) provide only predefined individual analysis facilities. They don't provide a way to specify application specific measurement, which should be done manually. To automate the application specific measurement, practitioners often use their own scripts, but they are designed case by case. There is no general frame study to our best knowledge.

DISCUSSION

In this study, a general frame study for automation of post-processing simulation results is proposed. The proposed frame study is practical as it provides flexibility, generality and ease of use.

It is flexible in that it can be easily extended to various purposes of analysis. There are commercial tools providing some of individual components in this frame study. For example, SOC Designer (http://www.arm.com) provides a facility to monitor bus utilizations. However, it does not provide any facility for detecting deadline misses.

The proposed frame study can be generally applied for any features of any applications. Some practitioners are using their own scripts that play like the manipulator. In most cases, the scripts are designed case by case.

The Boolean representation is simple to use but expressive enough to represent simulation results since complex semantics are handled by the manipulator not by the Boolean representation.

CONCLUSION

This study proposes a general framework of post-processing analysis on system behavior. By employing Boolean representation, the proposed technique can achieve flexibility, generality and ease of use, which are demonstrated with a case study.

REFERENCES

Clarke, E.M. and E.A. Emerson, 1981. Design and synthesis of synchronization skeletons using branching time temporal logic. Lecture Notes Comput. Sci., 5000: 196-215. DOI: 10.1007/978-3-540-69850-0_12

Corliss, M.L., E.C. Lewis and A. Roth, 2005. Low-overhead interactive debugging via dynamic instrumentation with DISE. Proceeding of the 11th International Symposium on High-Performance Computer Architecture, Feb. 12-16, IEEE Xplore Press, San Francisco, pp: 303-314. DOI: 10.1109/HPCA.2005.18

Drusinsky, D., M. Shing and K. Demir, 2005. Test-time, run-time and simulation-time temporal assertions in RSP. Proceeding of the International Workshop on Rapid System Prototyping. June 8-10, IEEE Xplore Press, Montreal, pp: 105-110. DOI: 10.1109/RSP.2005.50

Ghenassia, F., 2005. Transaction-Level Modeling with SystemC: TLM Concepts and Applications for Embedded Systems. 1st Edn., Springer, USA., ISBN: 0387262326, pp: 271

Metz, E., R. Lencevicius and T.F. Gonzalez, 2003. Performance data collection using a hybrid approach. ACM SIGSOFT Software Eng. Notes, 30: 126-135.

Open SystemC Initiative, 2005. Draft Standard SystemC Language reference manual version 2.1. Open SystemC Initiative (OSCI). http://www.cse.iitd.ernet.in/~panda/SYSTEMC/LangDocs/LRM_version2.1.pdf

Pnueli, A., 1977. The temporal logic of programs. In: Proceeding of the IEEE Symposium on the Foundations of Computer Science, Oct. 31 -Nov. 2, IEEE Xplore Press, Providence, RI., USA., pp: 46-57. DOI: 10.1109/SFCS.1977.32

Pinter, G. and I. Majzik, 2005. Automatic generation of executable assertions for runtime checking temporal requirements. Proceeding of the 9th International Symposium on High-Assurance Systems Engineering, Oct. 12-14, IEEE Xplore Press, Heidelberg, pp: 111-120. DOI: 10.1109/HASE.2005.6

Stewart, D.B. and G. Arora, 2003. A tool for analyzing and fine tuning the real-time properties of an embedded system. IEEE Trans. Software Eng., 29: 311-326. DOI: 10.1109/TSE.2003.1191796