SAAT: Reverse Engineering for Performance Analysis

Seon-Ah Lee, Seung-Mo Cho, Sung-Kwan Heo
Software Center, Corporate Technology Operations, Samsung Electronics Co. Ltd.
599-4, Shinsa-dong, Kangnam-gu,Seoul, Korea, 135-120
{salee, seungm.cho, sk.heo}@samsung.com

Abstract

It is essential to understand both the static and dynamic aspects of existing software for performance analysis. Software reverse engineering reestablishes the structure and behavior of software and helps with that understanding. Researchers in reverse engineering, however, have focused on identifying components and on static relationships. Efforts on performance engineering are being made to represent software behavior and simulate it. However, no one has tried to extract a simulated model from existing software automatically.

We introduce SAAT, a tool developed at our research center. SAAT analyzes the dynamic aspects of software and creates a simulated model for performance analysis. We explain how the model can be generated, using a case study of UPnP middleware. This paper contributes to the bridge between performance analysis and reverse engineering

1. Introduction

Performance analysis is a process that analyzes dynamic execution flow, estimates the time and resources consumed, discovers potential bottleneck points, and predicts the performance in a real environment. In order to analyze software performance, information for such analysis should be provided by software architecture models and design specifications. This information is required to help understanding and predict time-dependent behaviors during performance analysis by dividing software into modules and by displaying time, intercommunication, data access frequencies, data transfer capacity of communication channels and other data.

If the existing software's design specifications are incomplete or incompatible with the current software version, the design specifications may not be used in the performance analysis. Additionally, development team members are sometimes too busy to participate in performance improvement work. In that scenario, a reverse engineering methodology will analyze the performance of the software. However, past studies in reverse engineering have concentrated on static aspects, which extract relationships among components through source code analysis. To date, fields of performance analysis and reverse engineering have not been directly related.

In this paper, we introduce the SAAT tool that will analyze and represent the dynamic structures of software visually for performance analysis. In Section 2, we introduce previous studies for software performance analysis and dynamic reverse engineering. In Section 3, we explain the basic concepts of SAAT. In Section 4, we explain the technological considerations to implement and the architecture of SAAT. In Section 5, we present a sample case of UPnP middleware. In Section 6, we discuss our results up to this point and any remaining problems and recommend tasks for future study.

2. Previous Studies

Researchers in performance engineering are studying how to integrate software architecture with performance information. In the realm of reverse engineering, dynamic reverse engineering to extract software execution models from existing systems is also being tried. In this section, we discuss the progress of research in these two areas, sharing the common factor of software modeling. We will survey research related to the software performance model (2.1) and we will cover the reverse engineering research status for existing system analysis (2.2).

2.1. Software Performance Model

The software performance model enables one to measure the detailed performance of software. In addition, the performance model allows quick and convenient structural investigation when problems are found. To allow this solution, the performance model shall precisely describe the system to be improved. Related researches including the following:

Smith [3,4] pointed out that there is no software architecture specification documented enough for performance analysis in general, and proposed the PASA (Performance Assessment of Software Architecture)

methodology, which extracts architecture information from developer interviews and work products. PASA has 10 stages. In Stages 1 to 6, performance analysts examine software architecture and review the important use cases and scenarios with the development team. In Stages 7 to 10, the performance analysts construct and analyze the performance model, and announce the result. The PASA method requires dedicated cooperation from developers because the accuracy of the performance model depends on information provided by the developers.

Woodside [5,6] assumed that the contents that were not dealt with in software architecture documents, were omitted either because everyone understood the contents or they were something that didn't need to be described. He then presented the PASD (Performance Aware Software Development) methodology that produces and analyzes performance models from the design documents. The PASD has 7 stages. In Stages 1 to 3, performance-related information is added to the function-oriented specifications to make the specifications more complete. In Stages 4 to 5, the scenario model in the complete specification is transformed into a performance model. In Stages 6 to 7, performance is evaluated, and feedback is provided. In the PASD method, the performance model is created according to the specification's scenario model, and the accuracy of the specification affects the performance analysis.

Pooley [7] asserted that integration of performance factors with design methods shall precede the performance analysis framework and made efforts to integrate performance factors with UML notations. He also proposed simulation methods of the designmodels described in UML and performance analysis methods. The method presented by Pooley analyzes performance by producing simulation models with sequence diagrams, etc., used in dynamic modeling of UML or by changing using Petri-net models. Additionally, in Pooley's method, the accuracy of information given affects the performance analysis results of the model.

Similarly, researchers have made efforts to integrate the software performance model with software development methodologies and design models. If such efforts are connected with dynamic reverse engineering; more substantial effects can be achieved. First, it is possible to automate the creation of a performance model based on existing software. Accordingly, analysts might reduce time working with development team. In addition, the performance model does not need to rely on an incomplete design specification.

2.2. Software Reverse Engineering

In order to understand software, reverse engineering is used to identify software components and their interdependence and produces software design-level abstractions [8]. Software reverse engineering is being researched for various purposes, such as how to add new functions to existing software, maintain and improve system efficiency, and recycle modules in new systems. Recently, so-called dynamic reverse engineering has been started in an effort to discover software component interaction using software traces and records. The following discusses research related to the dynamic reverse engineering.

Systa[9,10] proposed the Shimba tool that automatically produces sequence diagrams of Java programs. With the Shimba tool, trace information is acquired while such programs are executed, and the information is then used to create a state diagram and a scenario diagram. Systa's papers give a lesson that dynamic aspects of software can be generated from monitoring software execution, but it does not propose to link the information to a performance model. Also, considering the fact that not many existing systems are constructed in Java, additional research is required for other languages.

Walker and Murphy [11,12] proposed an abstraction method, recognizing the fact that event trace information at the functional level presents a wide gap from the subsystem level of developers' interests. This method uses a visualization tool and a path query tool. The visualization tool shows a series of drawings according to system execution. The path query tool supports the analyze event flow information, using normal expressions that map the source codes to components of the developer's choice. The method presents the basic techniques in abstracting event trace information. However, the method seems to require some more time for field use, considering that it is limited to object-oriented languages, and no real application case has yet been presented.

Bengtsson and Bosch [13] pointed out that there was no research on architectural reengineering methodology, and if any, quality attributes were not considered. They defined a reengineering methodology based on scenarios. In this methodology, explicit and objective evaluation methods, such as simulations, mathematical model rings, etc., are adopted.

These efforts to produce architecture-level execution models from software execution flow have indeed begun [8,9,10,11,12,13]. The researches have presented many fundamental and useful results, but further efforts are required to make them practical such as expanding to the languages mainly used in real development. Also, there is still no attempt to connect the result to a simulation.

3. Concept of SAAT

The Software Architecture Analysis Tool was developed for performance analysis at the Software

Center, Samsung Electronics Co., Ltd. Since most of software programs are implemented in C language in Samsung electronics Co, Ltd., SAAT targets software constructed in C language. Our purpose was to overcome delays that accompanied performance analysis. Our activities of performance analysis are as follows: When a performance analyst is requested to analyze software performance, the analyst first has to understand the software's structure and its dynamic behaviors. Then, the analyst finds the component that unnecessarily consumes much time and resources. Finally, the analyst identifies improvement issues and solutions using a simulation tool. We hoped to shorten the time of the performance analysis in order to make the analyst's work more efficient. We tried to automate the analysis process.

The process of analyzing the software performance of the existing system at Samsung Electronics Co., Ltd., can be automated as shown in Figure 1. First, the information of how software modules interact should be recorded (Software Trace Data). Second, the interaction should be represented as nodes and edges in drawing a diagram (Behavior Model). If one would like to understand the dynamic structure of the software, a composite diagram explaining several interactions should be drawn (Execution Model). Last, the composite diagram should be converted for modeling in a simulation tool (Simulation Model). In the following subsections, we review the concepts of each model in Figure 1 in more detail.

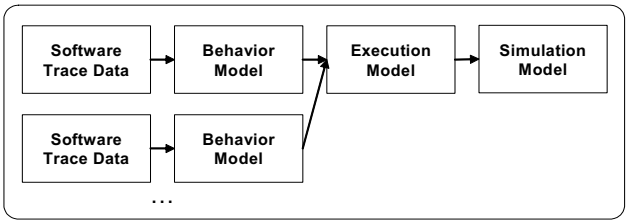

Figure 1: Automating process for performance analysis

3.1. Software Trace Data

Software Trace Data is a record of the interaction between the software modules. Because our target software is written in a procedural language, a software module is a function. Interaction involves not only call-relationships among several functions, but also information about the order of those functions. Hence, Software Trace Data includes function names, the calling relationship and execution order. Also, we add the function's running time into the data for performance analysis.

Software Trace Data is recorded as shown in Figure 2. The record consists of function name, time stamp, and flags according to the execution order. A line informing a function's start has information about function name, start time, and start flag. A line informing a function's end has information about function name, finish time, and finish flag. Also, lines of called functions are nested between the starting line and the finishing line of a calling function. As a result, both call-relationship and execution order can be represented in the Software Trace Data.

```
Function Main()
{
    call A();
    call B();
    call C();
}
Function A(parameter order)
{
    if (order == first_sequence)
    {
        call a();
        call b();
    }
    else
    {
        call c();
    }
}
Function B()
{}
Function C()
{}
```

```
A node  Time      Start/Finish

Main    start  time of Main  start  flag
  A     start  time of A     start  flag
    a   start  time of a     start  flag
    a   finish time of a     finish flag
    b   start  time of b     start  flag
    b   finish time of b     finish flag
  A     finish time of A     finish flag
  B     start  time of B     start  flag
  B     finish time of B     finish flag
  C     start  time of C     start  flag
  C     finish time of C     finish flag
Main    finish time of Main  finish flag
```

Figure 2: A concept of Software Trace Data

Software Trace Data is recorded while running the software. The easiest method for tracing software is to insert probing functions into functions of the target software. These probing functions write events of functions into a file when the functions start and finish. After compiling the software instrumented with probing functions, we can create a Software Trace Data file by running the software.

3.2. Behavior Model

The Behavior Model is a diagram representing Software Trace Data as nodes and edges. A node stands for a software module, namely, a function. Edges have two kinds of meaning. An edge directing below means the execution order between two connected functions. An edge directing to the right signifies a call-relationship between two linked functions. The Behavior Model is similar to a sequence diagram of UML, a design language for object-oriented programming. Thus, the Behavior Model represents one instance of the software executions.

Figure 3 shows an abstract presentation of the Behavior Model. Software modules are arranged according to their execution order and call relationship. In case "Main( )" calls "A( )," The Behavior model places "Main( )" left, "A( )" right, and shows an arrow starting from "Main( )" and arriving at "A( )." However, in another case, "Main( )" calls the second function, "B( )," Behavior model applies another rule: The second called function is connected to the first called function. Hence, the Behavior Model places "B( )" below "A( )" and shows an arrow starting from "A( )" to "B( )." The latter

arrow stands for the execution order between "A( )" and "B( )."

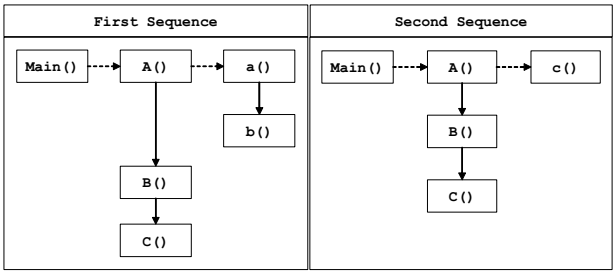

Figure 3: A concept of the Behavior Model

Rules of creating the Behavior Model from the Software Trace Data are related to start and finish flags. If a line has a start flag and the previous line also has a start flag, the node for the former is placed left of the node for the latter. Meanwhile, if a line has a start flag and the previous line has a finish flag, the node for the former is placed below the node for the latter. Additionally, if a line has a finish flag, there is no action for arrangement. The activity of arrangement occurs when reading a line including a start flag.

3.3. Execution Model

The Execution Model is a composite of several Behavior Models to represent the dynamic structure of the software. Even if each Behavior Model calls for the same function, the function may have different flows, based on control conditions and values of the variables. The Behavior Model can be combined by considering the control conditions, such as the branch or loop. To represent these control conditions, the Execution Model has several shapes of nodes. For example, the branch condition is represented as a rhomboid and the loop condition as a circle. We borrowed the concept of the Execution Model from the Execution Model of PASA[3].

The Execution Model can be extracted as shown in Figure 4 from the Behavior Models shown in Figure 3. A calling module nests called sub-modules. If a module has several different flows, the module possesses nodes showing the control conditions. For example, "A( )" has two different execution flows in Figure 3. Thus, "A( )" in Figure 4 has a branch node below the "Begin" node. After that, two different flows diverge from the branch node and converge to the "End" node in A( ).

The internal structure of functions may be extracted from former static reverse engineering tools. However, we have discovered that we can construct the Execution Model using only the information from the Behavior Models. While each Behavior Model is scanned, a branch node is inserted in case the Behavior Model has different part from the previous. When a function includes a repeated series of nodes, a loop node is added. Multiple Behavior Models are integrated to a single Execution Model in this way.

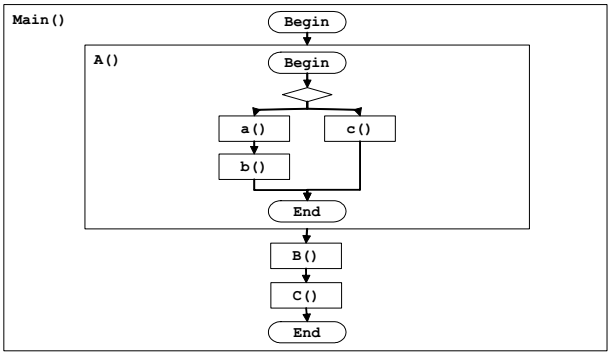

Figure 4: A concept of the Execution Model

3.4. Simulation Model

The Simulation Model is a model running in a simulation environment to demonstrate architectural issues of the present system. The Simulation Model is identical with the Execution Model in containing the information of software structure and performance information. Yet, we can modify, animate and simulate models in the simulation environment, so that we can predict a result of software implementation.

We can create the Simulation Model mapped from the Execution Model in figure 4 as follows. We delineate a model for "main ( )" in one module in our simulation tool. Next, we draw a model for "A( )" in a sub-model of the module separately. Finally, we create a transaction to trace models and insert a condition into the branch node in "A( )". After that, we can simulate the models.

We can create the Simulation Model automatically by mapping each element of the Execution Model to each element of the Simulation Model. The Execution Model is composed of graphs, sub-graphs, nodes, and edges. The Simulation Model is composed of modules, sub-models, nodes, and arcs. Therefore, graphs are mapped to modules; sub-graphs are mapped to sub-models; nodes are mapped to nodes; and edges are mapped to arcs. The arrow in the Execution Model is mapped to a transaction of the Simulation Model.

4. Implementation of SAAT

In this section, we present the Software Architecture Analysis Tool (SAAT) – a tool to generate a dynamic model for the performance analysis of software. We developed the prototype of SAAT to examine the feasibility of SAAT projects.

The architecture of SAAT may be drawn as the Figure 5. SAAT is related to existing commercial tools. TAU provides trace data extracted from software execution [14], while aiSee is a tool showing a model in GDL [15]. aiSee allows users to see the Behavior Model and Execution Model. Finally Workbench conducts Simulation Modeling [16]. SAAT is also composed of three parts: One that extracts the Behavior Model from trace data (BM); one that integrates several behavior models into an execution model (EM); and one that changes the Execution Model to the Simulation Model (SM).

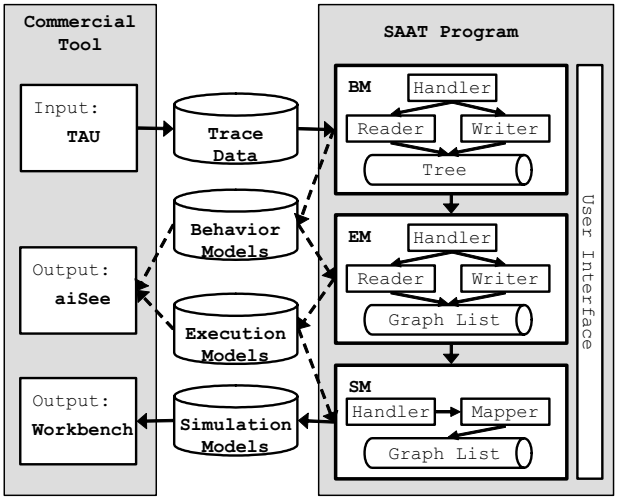

Figure 5: Structure of SAAT

4.1. Commercial Tools

4.1.1. TAU. The trace tool had to capture the system's response behavior to a user's request at regular intervals. Besides this basic role, we additionally had to consider multi-processors and multi-threads because current software consists of them. Fortunately, we found a reliable trace tool called TAU [14] for generating Software Trace Data. TAU supports multi-processors and multi-threads and generates data similar to the Software Trace Data in Figure 2.

4.1.2. aiSee. When we describe the Behavior Model and the Execution Model, we needed to determine a description tool. With respect to a description tool, we need a tool to present nodes and edges. Instead of implementing the tool, we searched for a convenient tool to present nodes and edges easily. We found that aiSee can present a diagram described in Graph Description Language (GDL). This language is a simple language to express graphs and has keywords like node, edge, graph, etc. [15].

4.1.3 Workbench. Since we use Workbench [16] as a simulation environment, we should convert the Execution Model to a model running in the Workbench. In the Workbench, one model is composed of nodes, arcs, and transactions. One model also could have several sub-models. By transforming the Execution Model into the simulation tool, we can obtain modeling data for simulating and analyzing our system's performance and verify its functionality.

4.2. SAAT Program

4.2.1 BM. When creating the Behavior Model, we should consider the abstraction level of the models so that users understand and manage the models easily. For that, the function lists should be categorized into modules to the level of the user's requirements. Also, low-level functions that users do not want to see should be eliminated or hidden to maintain the simplicity of the models. Thus, we endow a node possessing sub-nodes with a folding option to solve this issue. In spite of the importance of abstraction, the implementation was simple because aiSee supports the option with one token.

The drawing mechanism, used for the Behavior Model, is a binary tree. Each node in a binary tree has a left child and a right child. In the concept of Behavior Model, a left child becomes the first child, and a right childe turns to a sibling. The first child is the first called sub-function on the base of a function. A sibling implies the next executed function from a function. In reading, Reader saves information on the line to a node and pushes the node into a stack in temporary. When Reader reads a finish line related to the node in the stack, it pops the node and the previous node from the stack and links both to each other to make a tree. Then, Writer writes files by visiting from the root to leaf nodes in the tree.

4.2.2 EM. In creating the Execution Model, the internal structure of a function is needed because the Execution Model has a structure to fulfill the flows of several Behavior Models. For that purpose, if a function is called more than once and sub-operations of the function are different, loops and branches in the internal structure of the function need to be recognized. We replaced this structure of the function with the composition of the Behavior Models, as we wanted to get rid of cumbersome tasks like source analysis.

The structuring mechanism, used for the Execution Model, is a hash table that manages graphs that are organized in adjacency linked list. The hash key is a graph name. Once Reader reads tree information, Graph List recognizes nodes having a sub-tree, converts them to graphs, and saves the graphs in the hash table. After that process, Graph List scans the hash table. When Graph List finds the same name of a graph in the lists of graphs,

it combines the two graphs into one. Finally, Writer creates a file of the Execution Model by scanning the hash table. These tasks occur several times if there are more Behavior Models.

4.2.3 SM. When creating the Simulation Model in Workbench from the Execution Model in GDL, we should deliberate on the conversion from one language to another. One parser may be required if we intend to convert GDL to Simulation language in Workbench. For that task, we can use Lex and Yacc for analyzing GDL and converting the language to Workbench graph codes. However, we omitted the subpart because we used a hidden file to have the concise information of the graphs instead of using the GDL file directly. Mapper reads information on the graphs in a hidden file of the Execution Model and maps the information to the Simulation Model. Writer outputs the files to run in a simulation environment.

5. Case Study

We applied SAAT to Universal Plug and Play (UPnP) [17], developed by Samsung Electronics Co., Ltd. UPnP is a home network middleware that supports distributed and open networks that are used to control devices and transmit data among devices. UPnP consists of two components: Controlled Device (CD) and Control Point (CP). CD provides services, while CP detects and controls the services. The middleware is implemented in C language.

Our concern was the feasibility of SAAT. We wondered whether the tool could be applied to ongoing development projects and products and how effective the result of creating a simulation model would be. For that, we captured the Trace Data of UPnP, using the TAU tool (5.1) and generated Behavior Models (5.2), Execution Model (5.3) and Simulation Model (5.4) in order.

5.1. Trace Data of UPnP

We captured the Software Trace Data of UPnP as Figure 6. We used TAU in order to instrument source files of UPnP with probing functions. The result files were five files containing the information of each thread because UPnP runs in five threads and TAU generates trace files according to each thread. However, we just show one trace file here as an example.

The case in Figure 6 is an instance of a concept in Figure 2, Section 3.1. TAU logs the execution time as 16 digits. With regards to flags, 1 means a start flag and -1 signifies a finish flag. This Software Trace Data has the information on the execution flow of UPnP. We can know which executions occurred in what order by looking at the list of functions by time. However, for a more intuitive understanding, such execution flow must be graphically represented.

```
[root@duri93 device]# cat events.0.edf
#  creation program: tau_convert -dump
#  creation date: jul-08-2003
#  number records: 40
#  number processors: 0
#  max processor num: 0
#  first timestamp: 1057631241510426
#  last timestamp: 1057631247598443

#=NO= =====EVENT== ==TIME [us]= =NODE= =THRD= ==PARAMETER=
   1                  (null)          1057631241510426    0    0
   2                  (null)          1057631241510447    0    0
   3        "int main(void) C  "      1057631241510456    0    0        1
   4 "void TvDeviceStateTableInit(v   1057631241510563    0    0        1
   5 "void TvDeviceStateTableInit(v   1057631241510590    0    0       -1
   6 "int UPnP_CD_Start(int, FunPtr   1057631241510696    0    0        1
   7       "int upnpStart() C  "      1057631241510732    0    0        1
                                  .............
  29 "void UPnP_CD_SetRenewTime1057631242718027    0    0       -1
  30 "int UPnP_CD_Finish(void) C  "  1057631244597264    0    0        1
  31     "void Stop_Threads() C  "    1057631244597326    0    0        1
  32 "int PrintString(char *, ...)   1057631244597332    0    0        1
  33 "int PrintString(char *, ...)   1057631244597366    0    0       -1
  34 "int PrintString(char *, ...)   1057631247598364    0    0        1
  35 "int PrintString(char *, ...)   1057631247598410    0    0       -1
  36     "void Stop_Threads() C  "    1057631247598418    0    0       -1
  37 "int UPnP_CD_Finish(void) C  "  1057631247598423    0    0       -1
  38         "int main(void) C  "      1057631247598428    0    0       -1
  39                 (null)          1057631247598440    0    0
  40                 (null)          1057631247598443    0    0
```

Figure 6: Tautrace.0.0.0.trc (Main flow of UPnP CD)

5.2. Behavior Model of UPnP

The Behavior Model connected to the Trace Data in Figure 6 is shown in Figure 7. This Behavior Model shows the execution order from the Trace Data of TAU. We can easily know that UPnP CD starts, sets cache control, sets CD's timeout, sets renew time, and finishes by reviewing the diagram in Figure 7. Each node can be folded or unfolded to hide or show a sub-tree of each node. Therefore, users can browse the Behavior Model at the level they want to know.

Behavior Models differ by the user services requested. Figure 8 shows another Behavior Model of UPnP. The cases in Figures 7 and 8 are those of the Behavior Model explained in Section 3.2. In addition to the concept of Section 3.2, information on the time consumed at each node is displayed beside the node name. Thus we can know the candidates for any bottleneck as well as the execution order and call-relationship.

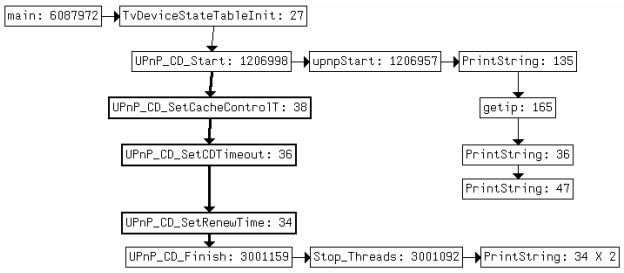

Figure 7: Behavior Model 1 (Main flow of UPnP CD)

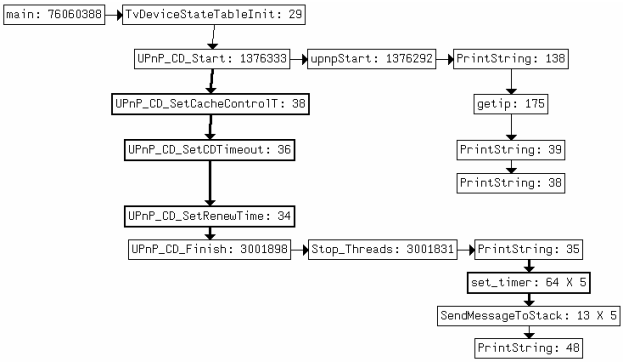

Figure 8: Behavior Model 2 (Main flow of UPnP CD)

5.3. Execution Model of UPnP

The Execution Model combines different Behavior Models. Figure 9 shows an Execution Model produced from the combination of Behavior Models of Figures 7 and 8 in Section 5.2. In Figure 9 below, the internal structure of "Stop_Threads( )" represents the Execution Model well. The "Stop_Threads ( )" presents a branch to two different internal flows and a loop showing repetition of the sub-flows.

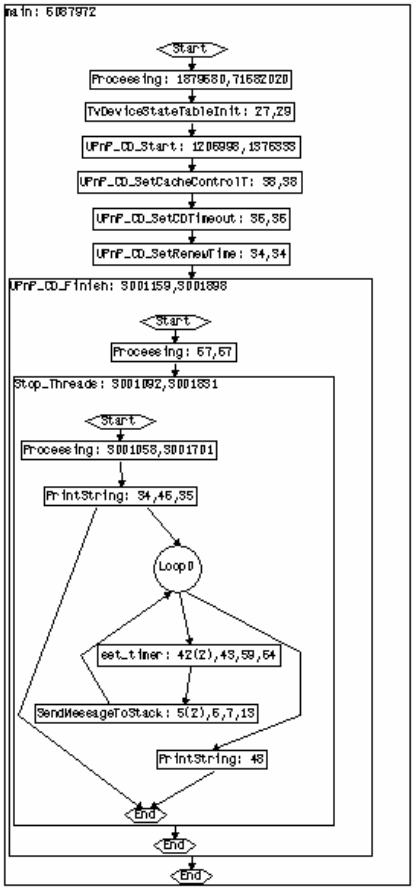

Figure 9: Execution Model (Main flow of UPnP CD)

5.4. Simulation Model of UPnP

The Execution Model in Figure 9 is changed to the Simulation Model in Figure 10. The upper diagram in Figure 10 shows a Workbench model for the main function while the lower diagram shows a model for Stop_ Threads( ). The distinctive aspect in the Simulation Model is the transactions, which move dynamically as time goes. Therefore, we can modify and simulate a model to predict the performance of the system to be implemented and search for a better solution based on the simulation results.

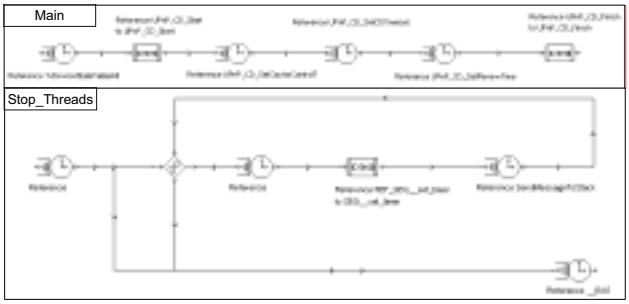

Figure 10: Simulation Model (Main flow of UPnP CD)

At this time, we confirm that the modeling data could be generated from monitoring software execution. However, the result of applying SAAT to the UPnP product is incomplete when comparing the result with that of a manual result for several reasons. First, SAAT pours all information into the Simulation Model, while performance analysts do not care for detailed layers. Second, the SAAT starts at the thread level while performance analysts start to model at the critical modules. Finally, SAAT does not distinguish concrete conditions, although some conditions could be important in modeling. We should develop SAAT to customize the Simulation Model according to user intention.

6. Conclusions

We have explained how we created the Execution Model from the execution trace data of the software system and how we constructed the Simulation Model from the Execution Model for performance analysis. Through SAAT prototyping, we showed that a simulation model could be automatically generated from the execution trace data of software. This case study shows the possibility of saving the time usually consumed in making a simulation model for performance analysis of software.

We propose the following additional research. First, we should find how to group functions that belong to the corresponding component. In this case, we may use the

Dali Workbench tool made by Kazman [18, 19]. However, we did not yet implement this method in SAAT. For this purpose, user intervention parts or important component-declaring parts must be added. In addition, we should complement SAAT by finding additional rules for converting from the Behavior Model to the Execution Model and by adding options to modify the Simulation model for user tastes. In the long term, we want to adapt this tool to several modeling environments.

7. References

[1] P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord, and J. Stafford, *Documenting Software Architecture: Views and Beyond*, Addison-Wesley, Sept. 2002.

[2] R.J. Pooley, "Software Engineering and Performance - a roadmap", *Proceedings of the conference on The future of Software engineering*, Limerick, pp189-200, July 2000.

[3] C.U. Smith and L.G. Williams, *Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software*, Addison-Wesley, Sept. 2001.

[4] C.U. Smith and L.G. Williams, "PASASM: An Architectural Approach to Fixing Software Problems", *Proc. CMG*, Reno, Dec., 2002.

[5] K. Siddiqui and M. Woodside, "Performance-Aware Software Development (PASD) Using Resource Demand Budgets", *Proc. of the 3rd WOSP*, Rome, July 2002.

[6] D. Petriu and M. Woodside, "Generating a Performance Model from a Design Specification", 3rd *Workshop on Generative Programming*, ECOOP 2001, June 2001.

[7] R. Pooley and P. King, "The Unified Modeling Language and Performance Engineering", *IEE Proceedings - Software*, Vol 146 No 1, pp 2-10, February 1999.

[8] E. Stroulia and T. Systä, "Dynamic Analysis For Reverse Engineering and Program Understanding", *Applied Computing Review*, ACM, vol 10, issue 1, 2002.

[9] T. Systä, "Understanding the Behavior of Java Programs", *Proc. of the 7th WCRE*, pp. 214-223, Brisbane, Australia, November 2000

[10] R. Kollmann, P. Selonen, E. Stroulia, T. Systä and A. Zündorf, "A Study on the Current State of the Art in Tool-Supporter UML-Based Static Reverse Engineering", *Proc. of the 7th WCRE*, pp.22-33, 2002.

[11] R. Walker, G. Murphy, J. Steinbok and M. Robillard, "Efficient Mapping of Software System Traces to Architecture Views", *CASCON*, 2000.

[12] R. Walker, G. Murphy, B. Free-Benson, D. Wright, D. Swanson and J. Isaak, "Visualizing Dynamic Software System Information through High-level Models", *Proc. of the 13th ACM SIGPLAN Conference on OOPSLA*, ACM Press, pp. 271-283, 1998.

[13] P. Bengtsson and J. Bosch, "Scenario-based Software Architecture Reengineering", *Proceedings of the 5th ICSR*, pp. 308-317, June 1998.

[14] University of Oregon, *TAU: Tuning and Analysis Utilities*, http://www.cs.uoregon.edu/research/paracomp/tau/, 1999.

[15] Absint, *aiSee*, http://www.aisee.com, 2002.

[16] *Workbench*, Hyperformmix, http://www.hyperformix.com/products/workbench.htm, 2003.

[17] Microsoft Corporation, *Universe Plug and Play Device Architecture*, http://www.upnp.org/download/UPnPDA10_20000613.htm, 2000.

[18] R. Kazman and S.J.Carriere, "Playing Detective: Reconstructing Software Architecture from Available Evidence", *CMU/SEI-97-TR-010*, Pittsburgh, PA: Software Engineering Institute, Canegie Mellon University, 1997.

[19] L. O'Brien and C. Stoermer, "Architecture Reconstruction Case Study", *CMU/SEI-2003-TN-008*, Pittsburgh, PA: Software Engineering Institute, Canegie Mellon University, 2003.