

UNDERSTANDING THE DYNAMICS OF INFORMATION SYSTEMS

Abdelwahab Hamou-Lhadj

Electrical and Computer Engineering, Concordia University, 1455 de Maisonneuve West, Montreal, Canada

Keywords: Software Engineering, Information Systems, Program Comprehension, Dynamic Analysis, Reverse Engineering.

Abstract: Information systems are in the process of undergoing significant transformations triggered by the Internet technology. However, most existing systems suffer from poor to non-existent documentation, which makes the maintenance process a daunting task even for a skilled software engineer. As a result, software engineers are often faced with the inevitable problem of understanding different aspects of the system before undertaking a simple maintenance task. This paper describes ongoing research in the area of program comprehension that aims at investigating efficient techniques for the understanding of the dynamics of software systems with a particular emphasis on information systems. The proposed approach is based on the analysis of system's execution traces. The long-term objective is to create effective tool support for software engineers working on maintenance tasks.

1 INTRODUCTION

Today's information systems are object-oriented, component-based, and distributed in multi-tier architectures. Maintaining such systems is often a complex task; software engineers must *understand* various aspects of a system before they can make changes that preserve reliability and other system attributes. The problem is further exacerbated by the fact that documentation of the system under study is rarely updated while key developers, knowledgeable of the system's design, commonly move to new projects or companies.

Understanding a software system requires both static and dynamic analysis techniques. The former focuses on exploring the structure of the system by analysing its source code whereas the latter provides insight into its behavioural properties. Both approaches aim to extract the system's components and their relations at different levels of abstraction.

Today's most prominent maintenance activities in the context of information systems consist of the migration of existing systems into Web technology, and the integration of the system's components using Web Services. Both activities necessitate the understanding of the way the system functions, i.e. software maintainers must understand the behavior

of the system before they can undertake the previously mentioned maintenance tasks.

In this paper, we describe ongoing research that focuses on techniques that permit the analysis of the dynamics of a software system. These techniques revolve around efficient analysis of execution traces.

Traces have the advantage of being precise and sensitive to the input data (Ball 1999). Unlike static analysis, where the analyst needs to go through the many different relationships of all the system artefacts, traces can be collected in such a way that they contain only the information needed to perform the maintenance activity at hand. In addition, system execution can be driven by specific input data which provides a powerful mechanism for relating program inputs, outputs, and behaviour.

Traces, however, have historically been difficult to work with since they may contain millions of events. There is a need to find ways to "shrink" their content while keeping as much of their essence as possible.

The proposed research aims at investigating how to best represent traces. The long-term objective is to integrate the results into reverse engineering tools that can be used by software maintainers to efficiently analyse the content of traces while performing maintenance tasks.

This paper is organised as follows: The next section discusses briefly the components of an execution trace. In Section 3, we present our research framework and the particular topics that are being investigated by our research team. In Section 4, we present related work. We conclude the paper in Section 5.

2 EXECUTION TRACES

Traces are often generated by executing the features of the system under study. Test cases have also been used at a less extent. There are different techniques for generating traces. The first is based on instrumenting the source code, i.e. inserting probes such as print-out statements at different locations in the source code. In the context of object-oriented systems, probes are usually inserted at each entry and optionally each exit of every method. Instrumentation is usually done automatically. The second approach consists of instrumenting the execution environment in which the system runs. For example, the Java Virtual Machine can be instrumented to generate events of interest. The advantage of this technique is that it does not require the modification of the source code. Finally, it is also possible to run the system under the control of a debugger. In this case, breakpoints are set at different locations (e.g. entry and exit of a method). This technique has the advantage of not modifying the source code and the environment; however, it can slow down considerably the execution of the system.

To reproduce the execution of a distributed object-oriented system, we need to collect the events related to object construction/destruction, method entry/exit, and process execution and synchronization (De Pauw 2002, Richner 2002). The latter usually requires a global clock in order to replay the execution of the system accurately.

There are various sorts of traces depending on the type of analysis performed. For example, traces of methods calls are often used to understand object interactions. Statement-level traces represent the information at a lower level of abstraction allowing software maintainers to detect potential defects in the system. Some researchers suggest using traces of inter-component interactions to depict the system's behaviour at the architectural level, e.g. (Walker 1998). The distributed and object-oriented nature of today's information systems might require the combination of multiple types of traces. One aspect of our research is to determine which traces are most suitable for the comprehension of the dynamics of information systems.

3 RESEARCH FRAMEWORK

Figure 1 depicts our approach for trace analysis. Traces are generated through code instrumentation. They are then visualized using a visualization environment. Due to the extraordinary size of typical traces, we need to develop simplification algorithms that can help software engineers explore the content of traces in an easy way. In addition, we would like to explore traces so as to recover behavioural design models that are usually lost as the system undergoes several ad-hoc maintenance tasks.

The bulk of the proposed research is three-fold that we present here and elaborate in more detail in the subsequent sections:

- **Trace-Simplification Techniques:** This research tackles the problem of reducing the complexity of traces while keeping as much of their essence as possible.
- **Design Recovery:** This work focuses on the recovery of high-level behavioural design models from traces.
- **Trace Visualization:** This work focuses on techniques for efficient representation of trace information.

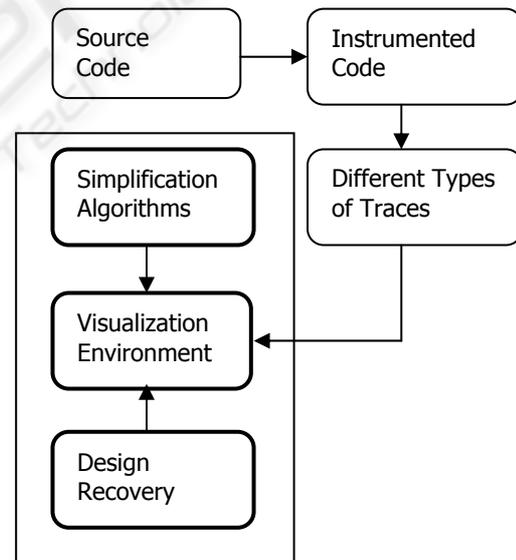


Figure 1: Framework of Proposed Research.

3.1 Trace Simplification Techniques

This work focuses on the problem of ‘compressing’ traces, thus enabling software engineers to better understand the behaviour of software. The term ‘compressing’ is not used here to refer to the well-known concept of data compression, but rather to

making a trace appear simpler and smaller so that software engineers can understand relevant parts more easily. The objective of the proposed project is to develop a number of algorithms for compression, particularly focusing on criteria by which various parts of a trace can be treated as the same pattern. For example, consider a portion of a trace of routine calls, T1: A(B(CCCCD)(B(DCC))), where A(B) denotes "A calls B". This trace can be transformed into T2: A(B(CD)) if the contiguous repetitions of "C" and the order of calls from "C" to "D" are ignored when comparing the two subtrees rooted at "B". At a high level, the information contained in T2 might be sufficient for the programmer's purposes.

Existing trace analysis tools, e.g. (Jerding 1997, Systä 2000, De Pauw 2002), support a variety of matching criteria that software engineers can use during the exploration of the trace. However, the sheer size of typical traces makes this exploration process a daunting task, further complicated by the fact that some criteria require, in advance, the setting of specific parameters. In addition, the order in which they are applied can have a significant impact on the resulting trace. Automated assistance is clearly needed. We propose developing a set of algorithms that will combine several criteria and automatically suggest appropriate settings for the rapid exploration of the trace content. The algorithms should be designed by taking into consideration the nature of the trace being studied (e.g. trace of routine calls, inter-process messages, etc.), as well as the current goals and experience of the maintainer. They will vary depending on the criteria used, the order in which they are applied, and input parameters specific to each criterion.

The proposed approach encompasses several steps. First, we need to conduct a comprehensive study of the most cited criteria in order to identify the ones that are best suited to the analysis of information systems. The study will involve applying these criteria to several large traces and using statistical techniques to analyse the results. Next, we will design and implement the algorithms starting with a few matching criteria. The remaining steps are performed iteratively: (1) Experiment with the algorithms using a multitude of traces generated from various information systems, (2) Validate the results by involving software engineers with different levels of knowledge of the system under study working on different maintenance tasks, (3) Refine the algorithms by modifying their input parameters, adding new matching criteria, etc, and finally going back to Step (1).

3.2 Design Recovery

The objective of this work is to develop efficient techniques for the recovery of high-level behavioural design views from execution traces. These views record the essence of traces in terms of a few abstracted elements, making it easier for an engineer to comprehend the information. In previous work, we introduced the concept of trace summarization (Hamou-Lhadj 2005) so as to extract summaries from large traces. The process relies on successive filtering of trace content by removing utilities. We worked with software engineers from the telecom. industry on developing a 'utilityhood' metric in order to assess, in the absence of proper documentation, the extent to which a component (e.g. class, method, etc) can be considered a utility (Hamou-Lhadj 2004, Hamou-Lhadj 2005). This metric is based on the idea that a component with higher *fan-in* is more likely to be a utility especially if the calls come from diverse parts of the system. In contrast, a component that is called from only a few places but calls many other components would most likely be an important component of the system.

The first step of the proposed project is to continue the work with the utilityhood metric in order to improve its effectiveness when applied to information systems. The second step is to experiment with the utilityhood concept so as to detect additional types of utility components, including processes, classes, packages, etc. The experiment will require the use of different target systems. The validation should involve the original designers of the systems if available or whatever other valid documentation is available. Furthermore, we need to experiment with traces generated from these systems and assess the accuracy of the high-level models extracted using the utility removal approach. The most important challenge we anticipate is to determine a proper utility threshold that can lead to views that are neither too abstract nor too detailed; in other words, views that are as informative as possible to software engineers.

3.3 Trace Visualization

The objective of this research is to develop a visualization environment for representing trace information using multiple views in order to provide effective support for program comprehension across a wide range of maintenance tasks. The main practical result we expect to achieve is a working tool that incorporates various views of the system's behaviour. The views will be linked to allow software engineers to navigate from one to another,

enabling them to analyse the system dynamics at different levels of abstraction.

The first step of this work will focus on determining the types of traces necessarily for the understanding of the behaviour of information systems. Due to the nature of today's information systems, we anticipate that the following views will be in use:

- **Call View:** A trace will be viewed as a call tree exhibiting the call relationship between the system components.
- **Process View:** This view will show a trace as a set of processes interacting by exchanging messages.
- **Object View:** This view will focus on object creation and deletion. It is particularly interesting for maintenance tasks that revolve around defect detection, performance analysis, etc.
- **Component View:** This view will allow software engineers to understand the interaction among the system components. This is important for information systems since they tend to be component-based.
- **Data View:** This view will focus on the way particular items in databases are updated.

In addition, the tool is expected to have supporting views. The main ones that we envision to be useful are: The Source Code View, which will be used to map trace elements to the source code, and the Statistics View, that will display statistical information to orient the user during the exploration of traces. The tool will allow the traditional browsing capabilities as well as the simplification algorithms discussed earlier. We anticipate building our tool as an Eclipse plug-in, so most of the tool infrastructure will be provided.

There are a number of key research challenges associated with this design. First, we need to have an internal model to represent the information displayed. This model must be scalable to handle lengthy traces and must have a sufficient power of expression to characterise the data generated from information systems. The second issue is related to the user interface widget that represents the traces. The problem is that most user interface elements for displaying large amounts of information build a complete representation of the display in memory, and then make sections of it visible as the user scrolls through the information. This is further complicated by the fact that when the user applies the compression algorithms, or simply changes some parameters, the entire display will need to be re-created; despite the fact that only a tiny fraction will be visible. In the context of this research, we will

investigate a new type of browsing widget that will generate the display for only that part of the trace that can currently be viewed. Furthermore, we will investigate the best way to represent traces in the user interface. We anticipate usability challenges related to the ability of each view to convey massive amounts of data. Research into software visualization and usability engineering will need to be carried out.

4 RELATED WORK

Existing trace simplification techniques can be grouped into four categories. The first focuses on grouping similar sequences of events invoked in a trace as instances of the same pattern (Jerding 1997, De Pauw 1998, Systä 2000, Richner 2002). Patterns are not easily exploitable unless generalized. A set of pattern matching criteria have been proposed by many authors, e.g. (Jerding 1997, De Pauw 1998). However, the use of these criteria has raised many research issues. One of the main objectives of the proposed research is to address these issues. The second category encompasses techniques that operate by limiting the amount of trace data gathered (Systä 2002). These techniques assume that software engineers have some knowledge of the system under study. This assumption is not valid in practice. Sampling techniques, which are representative of the third category, suggest that only a sample of the trace is needed for comprehension, eliminating the need to generate the entire trace (Chan 2003). Sampling is still at its early research stages. The main issue lies in determining appropriate sampling parameters. Finally, the last techniques suggest that trace simplification could be performed by clustering various trace components and only visualizing the interaction among these clusters (Walker 1998).

Another alternative to trace-simplification consists of generating summaries from large traces. Software engineers can use these high-level views to look at the big picture (i.e. main content) first and then delve into the detailed if desired. Amyot et al. suggest tagging the source code at particular places in order to generate a trace that can later be represented using a Use Case Map (Amyot 2002). Systä proposes a semi-automatic technique in which state diagrams can be synthesized using various UML sequence diagrams, extracted from traces (Systä 2002). Her approach combines static and dynamic analysis techniques. Wilde et al. propose a simple method for the recovery of design threads from inter-process systems by identifying, using

dynamic analysis, the implementation components relevant to each thread (Wilde 1997).

We have previously published a survey of the many existing trace visualization tools; this included descriptions of their advantages and limitations (Hamou-Lhadj 2004). These tools support features ranging from simple trace exploration techniques to more sophisticated types of analysis (e.g. querying of trace model, etc). Traces have been represented in various ways depending on their type. Traces of routine (method) calls are often visualized using tree structures (De Pauw 1998), UML sequence diagrams (Jerding 1997, Systä 2000, Richner 2002), and Use Case Maps (Amyot 2002). Traces of inter-process messages are usually represented using crossing lines among the processes (De Pauw 2002). Tools that support traces of architectural components use boxes and lines to represent the components and their dynamic interactions (Walker 1998).

5 CONCLUSIONS

In this paper, we discussed ongoing research in the area of reverse engineering of software systems with an emphasis on information systems. Our approach is based on analysing the content of large traces.

Traces, however, can be extremely large. Therefore, there is a need to investigate ways to reduce their size and complexity while keeping as much of their essence as possible. We discussed three research topics that we are currently investigating. The first one focuses on shrinking traces by grouping various sequences as instances of the same pattern. The key challenges consist of finding the proper matching criteria as a measure of similarity. We proposed developing a set of simplification algorithms based on these criteria.

Design recovery techniques, which are representative of the second category, focus on recovering behavioural design models from large traces. These models can be used by software engineers to explore a trace by looking at the main content first and then dig into the details.

Finally, the last research topic focuses on developing a visualization environment for representing traces. The environment should support multiple views so as to allow software engineers browse the content of traces at different levels of abstraction.

REFERENCES

- Amyot, D., Mussbacher, G., and Mansurov, N., 2002. Understanding Existing Software with Use Case Map Scenarios. In *SAM'02, 3rd SDL and MSC Workshop*, LNCS Vol.2599, Springer-Verlag.
- Ball T, 1999. The Concept of Dynamic Analysis. In *ESEC'99, 7th European Software Engineering Conference*, Springer-Verlag.
- Chan A., Holmes R., Murphy G. C., and Ying A. T. T., 2003. Scaling an Object-Oriented System Execution Visualizer through Sampling. In *IWPC'03, 11th International Workshop on Program Comprehension*, IEEE Computer Society.
- De Pauw W., Lorenz D., Vlissides J., and Wegman M., 1998. Execution Patterns in Object-Oriented Visualization. In *USENIX'98, 4th Conference on Object-Oriented Technologies and Systems*.
- De Pauw W., Jensen E., Mitchell N., Sevitsky G., and Vlissides J., Yang J., 2002. Visualizing the Execution of Java Programs. In *LNCS Vol. 2269*, Springer-Verlag.
- Hamou-Lhadj, A., Braun, E., Amyot, D and Lethbridge, T.C., 2005. Recovering Behavioral Design Models from Execution Traces. In *CSMR'05, 9th European Conference on Software Maintenance and Reengineering*, IEEE Computer Society.
- Hamou-Lhadj, A., and Lethbridge T., 2004. A Survey of Trace Exploration Tools and Techniques. In *CASCON'04, 14th Annual IBM Centers for Advanced Studies Conferences*, IBM Press.
- Hamou-Lhadj, A., and Lethbridge, T.C., 2004. Reasoning About the Concept of Utilities. In *ECOOP-PPPL'04, 1st International Workshop on Practical Problems of Programming in the Large*, LNCS Vol 3344, Springer-Verlag.
- Jerding D., Stasko J. and Ball T., 1997. Visualizing Interactions in Program Executions. In *ICSE'97, 19th International Conference on Software Engineering*, ACM Press.
- Richner T. and Ducasse S., 2002. Using Dynamic Information for the Iterative Recovery of Collaborations and Roles. In *ICSM'02, 18th International Conference on Software Maintenance*, IEEE Computer Society.
- Systä T., 2000. Understanding the Behaviour of Java Programs. In *WCRE'00, 7th Working Conference on Reverse Engineering*, IEEE Computer Society.
- Walker R. J., Murphy G. C., Freeman-Benson B., Swanson D., and Isaak J., 1998. Visualizing Dynamic Software System Information through High-level Models. In *OOPSLA'98, 13th Object-Oriented Programming Systems, Languages, and Applications*, ACM Press.
- Wilde N., Casey C., Vandeville J., Trio G., Hotz D., 1997. Reverse Engineering of Software Threads: A Design Recovery Technique for Large Multi-Process. *The Journal of Systems and Software*, Elsevier.