In the Proceedings of the 2002 WCRE Workshop on Software Architecture Reconstruction, pp. 5-7.

Using Log Files to Reconstruct State-Based Software Architectures

Gerald C. Gannod*† and Shilpa Murthy
Dept. of Computer Science & Engineering
Arizona State University
Box 875406
Tempe, AZ 85287-5406
E-mail: {gannod,smurthy}@asu.edu

Abstract

A common activity that occurs within the software development community is the use of log files to generate traces of observed software behavior. As a resource for reverse engineering, a log file has the advantage of being an accurate account of software behavior. Model checking approaches work by using exploration to determine whether certain temporal and safety conditions exist within the state space of some state-based model. In this paper we describe an approach to reverse engineering that combines the use of model checking and log file analysis to reconstruct software architectures.

1 Introduction

Software reverse engineering is defined to be a process of analyzing software components and their interrelationships in order obtain a description of the software at a high-level of abstraction [1]. Approaches for reverse engineering have been suggested that address the reverse engineering problem from a wide variety of views ranging from structural makeup to observable behaviors using a wide variety of techniques covering both static and dynamic analysis.

A common activity that occurs within the software development community is the use of log files to generate traces of observed software behavior. Several different approaches for creating log files exist including the use of compiler directives at the time of development and post-development instrumentation [2].

As a resource for reverse engineering, a log file has the advantage of being an accurate account of software behavior. However, a disadvantage is that the log file potentially provides only a subset of possible behaviors of the software. As a result, to mitigate the risk of using log files as a source of information for whatever reason, approaches such as adequate testing [3] are needed to ensure that a log file trace provides a reasonable or adequate amount of behavioral coverage.

Model checking approaches work by using exploration to determine whether certain temporal and safety conditions exist within the state space of some state-based model. Model checking, using tools such as Spin and SMV, has gained much attention recently due to many factors including the fact that it is relatively lightweight when compared to other formal approaches such as theorem proving.

In this paper we describe an approach to reverse engineering that combines the use of model checking and log file analysis to reconstruct software architectures. The remainder of this paper is organized as follows. Our approach to reverse engineering is presented in Section 2. Section 3 presents related investigations and Section 4 draws conclusions and outlines future investigations.

2 Approach

This section describes the underlying conceptual basis as well as the reverse engineering approach being used to reconstruct software architectures.

2.1 Underlying Conceptual Basis

Figure 1 depicts the general context for a software development approach that involves the use of model development, model checking, and log file generation (or other debugging approaches). In the general context, the relationship between models and programs is labeled as *implements-models*. The semantics of this relationship emphasizes the fact that the model is implemented by a program and that a program is modeled by a model.

Model checking tools such as Spin [4] facilitate verification of the ordering of events within a model. In Spin the verification is achieved using linear temporal logic (LTL) while SMV [5] achieves this using computational tree logic (CTL). Here we assume the use of Spin and LTL. The *maps-to* relationship between log files and LTL specifications emphasizes the analogy between log files and LTL specifications. As with the relationship between models

*Contact Author.

†This author supported in part by NSF CAREER Grant CCR-0133956.

and programs, there is a difference in the level of abstraction between log files and LTL specifications.

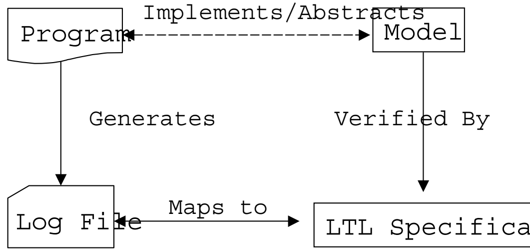
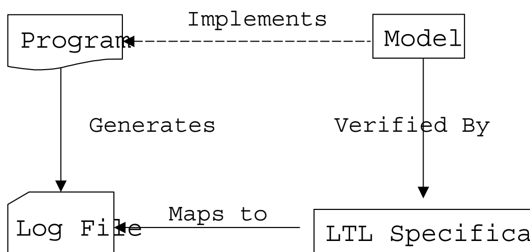
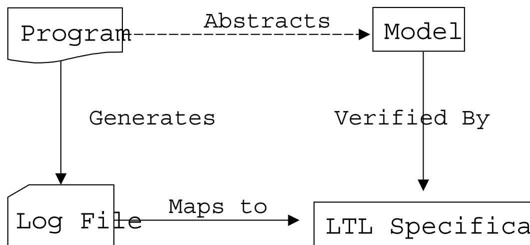


Figure 1. General Context

Andrews' developed an approach to software testing based on the concept that given a state-based model such as a finite automata or statechart, a program implementing the model must generate behaviors that correspond directly to events in the model [6]. While the approach does not use a specific model checking environment like Spin or SMV, the employed verification technique used does achieve similar goals, e.g., verification of the model by using log file events to drive model execution. As a result, a situation arises as shown in Figure 2(a), where a model is used to aid in the development of a program, and a log file is generated during program execution. In this case, the model is assumed to be correct and thus the verification against an LTL specification ignored. If the log file events do not correspond to model events, then the program is considered faulty and must be modified.



(a)



(b)

Figure 2. (a) Testing Context (b) Reverse Engineering Context

An interesting result can be derived if the relationships between program/model and logfile/LTL specification are reversed as shown in Figure 2(b). Namely, it provides a means for verifying correspondence between existing systems and models reconstructed using reverse engineering. That is, assuming that the program is correct (e.g., we are interested in learning what the program does rather than verifying it against requirements) then any log file produced is an accurate representation of what happens during execution. Therefore, any model constructed to represent the program behavior must at some level generate the events found in a log file with the same temporal orderings, assuming a well-defined logging policy.

2.2 Process

We are currently developing a reverse engineering approach based on the context provided in Figure 2(b). The approach has four primary steps that are necessary to be successful. In Step 1, log files must be generated from the program. This step, in some cases, is already completed since a common activity for some developers is to generate traces as an aid to debugging. However, there is a risk in using such traces since the logs can be inconsistently updated (e.g., not all events are captured). As a result, it is necessary to develop logging tools that follow specific logging policies on *what to log* and *when to log*.

In Step 2, a model must be reconstructed from source code and other sources of information. In our technique, we assume that the system of study utilizes a state-based architectural style and that we are interested in deriving high-level state charts as well as some intermediate models. Several approaches have been suggested for deriving state models from code and other artifacts including the work by Systa [7]. In our approach, model construction can be free form in the sense that it does not have to be strictly bottom-up. In fact, the model can be constructed top-down with little or no direct code analysis. As long as the verification step ends in a valid model, the approach used for deriving the model is unimportant.

Step 3 involves the mapping of log file events to LTL or CTL specifications, with the choice depending on the tool used to perform model checking. The mapping of log file events to LTL propositions requires the use of event abstraction techniques [9] especially since many events in a log file may be decompositions of some higher level event that is captured in a model. We are developing an approach that will work by partitioning log files into disjoint equivalence classes of events and mapping a representative of a partition to specific events contained in a model. The result preserves ordering while relaxing a need for exact event mapping.

Finally, in Step 4, model checking is used to determine whether the sequence of events captured in a log file and encoded with a LTL specification are consistent with the candidate models developed in Step 2. When an invalid verification occurs the conclusion that can be drawn is that either the model is incorrect, the encoding of the LTL spec-

ification is incorrect, or both. In the case of an incorrect model, modification and refinement of the model becomes necessary. In regards to the correctness of an LTL specification, as long as the generated specification preserves ordering located in the log file *and* the propositions correspond to events in the model, the verification will provide accurate results.

Upon completing the four steps, what an analyst would gain is a state-based architectural and design view of a system that has been verified against observable behavior exhibited by the system of study. Using the evaluation dimensions defined by Gannod and Cheng [8], the abstraction *distance* and *traceability* of the model from the original source code is influenced entirely by the approach used to perform Step 2 rather than by the underlying reverse engineering framework described here. In terms of *accuracy*, the verification step ensures that the model is accurate with respect to generated log files. However, the completeness of the verification depends on adequacy criteria used to determine how many log files to generate. The *precision* of the approach is considered to be high since the representation used to encode the models must be formal enough to employ model checking techniques.

3 Related Work

Andrews [6] provides a framework for automatically analyzing log files using state machines in the context of testing. The Log Files are created using logging policies and a standard format which specifies certain keywords. A log file analyzer is specified formally using the *Log File Analysis Language* (LFAL) and is built as a set of parallel state machines with each log file machine checking one thread of event.

Basten investigated the use of event abstraction to reduce the apparent complexity of distributed computations [9]. The work is based on the study of four aspects of event abstraction: a model describing primitive behavior, a formalism for specifying abstract behavior in terms of activity and causality, abstract descriptions of behavior, and verification of primitive or abstract descriptions against specified behavior.

Systa [7] discusses an experimental environment for reverse engineering Java software with respect to the concepts of static and dynamic views. These views contain overlapping information about software artifacts and their relations, which form a connection for information exchange between the views. Static information extracted from Java class files is viewed using the Rigi environment [10]. The dynamic information generated by running the software under a debugger is viewed as scenario diagrams using the SCED prototype tool. SCED state diagrams can be synthesized from these scenario diagrams. The SCED scenario diagrams are further used for slicing the Rigi view and the Rigi view in turn is used to guide the generation of SCED scenario diagrams and for raising their level of abstraction.

4 Conclusions and Future Investigations

Our initial experience with the approach has shown a great deal of promise with some moderate risks in each of the process steps. An ability to capture appropriate event information (Step 1) and map those events (Step 3) are especially crucial to the success of the approach since inaccuracies in these steps reduce the impact of the verification step.

To date, we have completed the construction of a logging tool that is based on the use of the Java Debbugging Platform [11] in order to log events in Java programs without resorting to behavior modification as is possible with code instrumentation techniques. We are currently developing tools to support the derivation of LTL specifications from log files using event abstraction techniques [9].

References

[1] Elliot J. Chikofsky and James H. Cross. Reverse Engineering and Design Recovery: A Taxonomy. *IEEE Software*, 7(1):13–17, January 1990.

[2] Kevin Templer and Clinton L. Jeffery. A Configurable Automatic Instrumentation Tool for ANSI C. In *Proc. of the Automated Software Engineering Conference*, 1998.

[3] Elaine J. Weyuker. The Evaluation of Program-Based Software Test Data Adequacy Criteria. *Communications of the ACM*, 31(6):668–675, June 1988.

[4] Gerard Holzmann. The Spin Model Checker. *Transactions on Software Engineering*, 23(5):279–295, May 1997.

[5] K. L. McMillan. Getting started with smv. Cadence Berkeley Labs, March 1999.

[6] James H. Andrews. Testing using Log File Analysis: Tools, Methods and Issues. In *Proc. of 13th Annual International Conference on Automated Software Engineering (ASE'98)*, pages 157–166, Honolulu, Hawaii, October 1998.

[7] Tarja Systa. On the Relationship between Static and Dynamic Models in Reverse Engineering Java Software. In *Proc. of the 6th Working Conf. on Reverse Engineering (WCRE99)*, pages 304–313, 1999.

[8] Gerald C. Gannod and Betty H. C. Cheng. A Framework for Classifying and Comparing Software Reverse Engineering and Design Recovery Techniques. In *Proc. of the 6th Working Conf. on Reverse Engineering*. IEEE, October 1999.

[9] Twan Basten. Event Abstraction in Modeling Distributed Computations. In *K. Ecker and M. Krmer, editors, Workshop on Parallel Processing, Proc.*, pages 46–65, Lessach, Austria, September 1993.

[10] Scott R. Tilley, Kenney Wong, Margaret-Anne Storey, and Hausi A. Müller. Programmable Reverse Engineering. *The International Journal of Software Engineering and Knowledge Engineering*, 4(4):501–520, 1994.

[11] The Java Platform Debugger Architecture(jpda). [Online Available] http://java.sun.com/products/jpda.