

Verification of Recovered 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 technique employed by software developers 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 safety and liveness conditions are satisfied by a finite-state model. In this paper we describe an approach that combines the use of model checking and log file analysis to facilitate verification of recovered models.

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]. Suggested approaches to reverse engineering include the use of structural remodularization and architecture recovery [2]. These techniques are primarily static, although some dynamic approaches have been developed [3]. Once a model is obtained, a question of traceability arises with regards to consistency between the original source and the derived model. As the gap between the levels of abstraction of an original source and reverse engineered model increases, this question of consistency becomes increasingly important.

A common technique employed by software developers 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 [4]. 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 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 static behavior sampling [5] 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 finite-state model. Model checking, using tools such as Spin [6], 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 that combines the use of model checking and log files to facilitate the analysis of the consistency between evidence obtained from an executable system and a model reconstructed from source code. As such, this paper suggests a technique for achieving a level of synergy between static and dynamic analysis. The remainder of this paper is organized as follows. Background material is presented in Section 2. The verification framework is introduced in Section 3. Strategies for architecture reconstruction and their relationship to the proposed architecture recovery verification framework are discussed in Section 4. A short example demonstrating the approach is described in Section 5, while Section 6 presents related work. Section 7 draws conclusions and suggests future investigations.

2. Background

In this paper, we use model checking technology as the means for verifying the consistency between log files and reverse engineered models. Model checkers are systems that have been used to study and verify behaviors of reactive and concurrent systems. Use of model checkers typically involves the creation of finite-state models that describe expected system behavior. The model checker itself uses state space exploration methods to determine whether some property is exhibited by the system. Model checkers are considered “pushbutton” systems since the output of the tools simply provide a “yes” or “no” answer, with the “no” often being accompanied by a counterexample. Properties

*This research supported in part by NASA Langley Grant NAG-1-2241 and NASA IV & V Grant NAG-5-12584.

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

[‡]Contact Author.

that are typically verified in the models are safety properties (expressed as state invariants) or liveness properties (expressed as a linear temporal logic (LTL) or computational tree logic (CTL) equations).

In this paper, we use the Spin [6] model checker in the examples to demonstrate the framework. Spin [6] has been used to verify the behavior of a wide variety of hardware and software systems. Promela, the input language for Spin, is based on the Dijkstra guarded command language as well as CSP.

To facilitate analysis, Spin utilizes both simulation and model checking via state space exploration. The simulation component of Spin supports the construction of message sequence charts as well as features that allow a user to observe various run-time properties of models including values of data variables, and step pointers that indicate the current program counter.

3. Approach

This section describes the underlying conceptual basis of a framework for verifying the consistency of reconstructed or recovered software architectures with dynamic information obtained from log files and traces.

3.1. Underlying conceptual basis

Figure 1 depicts a general context for software development. In this context, models are developed and verified using model checking against some set of LTL specifications. Additionally, programs are created using the model as a specification of behavior, and log files or other debugging approaches are used to understand and test the program.

In the general context, the relationship between models and programs is labeled as *implemented by-abstracted by*. The semantics of this relationship emphasizes the fact that the model is implemented by a program and that a program is abstracted by a model. The *maps-to* relationship between log files and LTL specifications emphasizes the analogy between log files and LTL specifications. Specifically, trace behavior of a program are analogous to LTL verifications of a model.

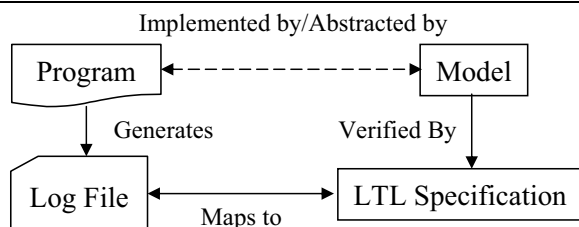


Figure 1. General context

Andrews [7] developed an approach to software testing based on the concept that, given a finite-state model such as a finite automata or statechart, a program implementing

the model must generate behaviors that correspond directly to events in the model. While Andrews' approach does not use a specific model checking environment like Spin, the employed verification technique achieves similar goals, e.g., verification of a model by using log file events to drive model execution. Consequently, 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. As such, any verification property exhibited by the model must also be exhibited by the program (and observable via some event trace). If the log file events do not correspond to model events, then the program is considered faulty and must be modified.

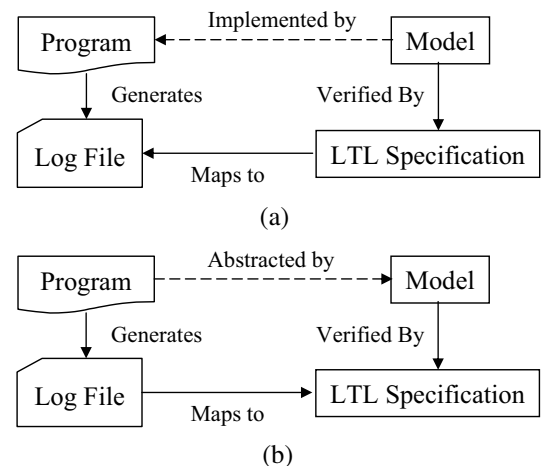


Figure 2. (a) Testing context (b) Reverse engineering context

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.

Figure 3 contains a log file generated by a program that is used to control a heating system based on temperature sensor data. This log file represents output that is commonly captured from systems either as debugging output or via event capture systems. In this case, the data was captured using an event capture system that has been developed using the Java Platform Debugger Architecture [8].

Once a log file like the one shown in Figure 3 is generated, it can be used to verify the correctness of a recovered

```

1.  hhs.components.Sensor.getTemp()
2.  hhs.Controller STATUS: heaterStat = 1
3.  hhs.components.Burner.status()
4.  hhs.components.Burner STATUS: STAT = ON
5.  hhs.components.Pump.status()
6.  hhs.components.Pump STATUS: STAT = ON
7.  hhs.components.Sensor.getTemp()
8.  hhs.components.Sensor.setTemp()
9.  hhs.components.Sensor STATUS: temp = 13
10. hhs.components.Sensor.setTemp()
11. hhs.components.Sensor STATUS: temp = 14
12. hhs.components.Sensor.setTemp()
13. hhs.components.Sensor STATUS: temp = 15
14. hhs.components.Sensor.setTemp()
15. hhs.components.Sensor STATUS: temp = 16
16. hhs.components.Sensor.getTemp()
17. hhs.components.Sensor.setTemp()
18. hhs.components.Sensor STATUS: temp = 17
19. hhs.components.Sensor.setTemp()
20. hhs.components.Sensor STATUS: temp = 18
21. hhs.components.Sensor.setTemp()
22. hhs.components.Sensor STATUS: temp = 19
23. hhs.components.Sensor.setTemp()
24. hhs.components.Sensor STATUS: temp = 20
25. hhs.components.Sensor.getTemp()
26. hhs.Controller STATUS: heaterStat = 0

```

Figure 3. Example log file

model of an architecture, as long as the architecture is expressed as a state model (See Section 4 for further discussion on this issue). In order to use a log file in this manner, its contents (or subsets thereof) must be translated into an equivalent logical form.

Dwyer et al. [9] introduced the notion of specification patterns for finite-state verification as a means for specifying and verifying common properties. Properties found in log files can be easily described using many of the patterns described by Dwyer et al. For instance, the *precedes* pattern captures the case where one event occurs before another. Specifically, to express that event *S* precedes event *P* between the scope of when events *Q* and *R* occur, we can write an LTL specification as follows:

$$\Box((Q \wedge \Diamond R) \Rightarrow (\neg P \ U \ (S \vee R))) \quad (1)$$

Where \wedge , \Rightarrow , \neg and \vee have the usual meaning, and \Box , \Diamond and U represent *everywhere*, *eventually* and *until*, respectively. The expression in Equation 1 explicitly states that within the scope of events *Q* and *R*, where it is assumed that *Q* happens sometime before *R*, the event *S* precedes event *P*. Consider the log file in Figure 3, where lines 2 and 26 indicate the assignment of a variable called *heaterStat* to 1 and 0, respectively. Additionally, lines 4 and 6 respectively indicate the turning on of a burner and pump. Mapping lines 2 and 26 to the scope events *Q* and *R*, and lines 4 and 6 to the precedence events *S* and *P*, Equation 1 can be rewritten as:

$$\Box(((\text{heaterStat} = 1) \wedge \Diamond(\text{heaterStat} = 0)) \Rightarrow (\neg \text{PumpOn} \ U \ (\text{BurnerOn} \vee (\text{heaterStat} = 0)))) \quad (2)$$

Which indicates that the burner is turned on before the pump between the scope of *heaterStat* becoming 1 and 0. Once a finite-state model has been recovered from code or other software artifact for the home heating system that generated the log file from Figure 3, Equation 2 can be used to perform verification using a model checker. The availability of log files along with appropriate log file generation strategies, provide a rich source of data with which to verify the correctness of derived architectural models, thus facilitating increased confidence regarding the accuracy of an applied reverse engineering technique.

Analysis of a reverse engineered model, can occur in two steps; a simulation step and a model checking step. A simulation run can be executed in Spin, which produces a message sequence chart similar to the one found in Figure 4. As the annotations point out, the message sequence chart depicts the same general scenario found in the log file and represents, graphically, the property found in Equation 2. Specifically it shows the relationship between the starting the heater (e.g., *heaterStat* = 0) and turning the burner and pump on. In the diagram, the vertical lines correspond to a lifeline for a process. Arcs between rectangles represent messages between processes. The model checking step can be executed using LTL specifications such as the one found in Equation 2.

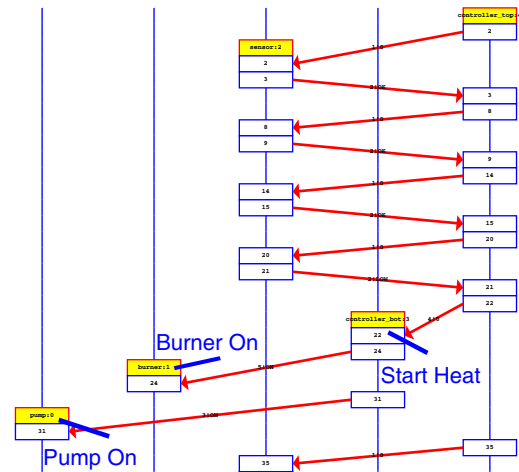


Figure 4. Spin message sequence chart

3.2. Process

We are currently developing a reverse engineering verification approach based on the context provided in Figure 2(b). The approach has four primary steps. In Step 1, log files must be generated from the program. This step, in some cases, is already completed since a common activity for software 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 log-

ging tools that follow specific logging policies on *what to log* and *when to log*. To assist in this endeavor, approaches such as those suggested by Mittermier and Pozewaunig [5] can be used to provide a proper scope of scenarios in which to collect traces.

In Step 2, a model must be reconstructed from source code and other sources of information. Section 4 directly addresses this issue. From a strategic standpoint, the selection of a particular model reconstruction approach depends upon the goals of the recovery activity. For instance, if an analyst wishes to verify that a system exhibits behaviors indicative of a particular architectural style, an approach tailored towards verifying such behaviors may be called for.

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 activity depends heavily upon the level of verification that is desired and, just as in the case of model construction, relies upon the goals of the overall recovery activity.

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 specification 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.

Steps 1 and 4 are currently supported by tools. An event capture system developed using Java Platform Debug Architecture is used for Step 1 and the model checking tool Spin is used to perform Step 4. Step 2 relies on the multitude of approaches available from the reverse engineering community. Support for Step 3 is currently under development.

4. Strategies

This section discusses a number of strategies for recovering software architectures from software systems and the relationship of those strategies to the verification methodology described in the previous section.

4.1. Overview

As stated earlier, the approach for verifying a recovered software architecture is free form in the sense that the verification method is not tied to any specific reverse engineering approach. However, it is necessary to have a log file that captures system events.

Scalability of the approach is defined in terms of reverse engineering strategies used to recover system architecture

and design. Since various reverse engineering approaches are already available, scalability depends on the employed reverse engineering technique. That is, the approach described in this paper addresses verification of architectures produced by the currently available reverse engineering approaches.

There are several strategies that can be supported by the verification approach described in this paper including forming a guess or *hypothesis*, deriving a model from a log of events (i.e., *event abstraction*), reconstruction of *stateless* structural models, or reconstruction of *stateful* behavioral models. These strategies may, of course, involve varying levels of human intervention.

Table 1 summarizes the strategies most often supported by current reverse engineering and software architecture recovery approaches. The “+” symbol indicates those strategies that the verification approach described in this paper is able to support, while the “-” symbol indicates strategies that cannot be supported (for the obvious reason that if source code is not available, models cannot be derived from that source).

Table 1. Supportable recovery strategies

Strategy	Code	
	With	Without
Hypothesis	+	+
Event Abstraction (from log)	+	+
Stateless (from code)	+	-
Stateful (from code)	+	-

4.2. Hypothesis-based abstraction

Hypothesis-based methods are those that are based on forming a guess or hypothesis regarding the architecture of a system. An associated verification task in this case would set out to verify the hypothesis using events in a log file as well as a model representing the guess. The difficulty of such an approach lies in producing the model that serves as the guess. Without any artifacts to base the guess on other than experience using or maintaining the subject system, reaction to the failure of the verification step becomes much more problematic since there is little information available to assist in modifying the hypothesized model. Figure 5 depicts the verification framework in the context of a hypothesis-based approach. In this context, a guess or hypothesis is used to translate the program into an appropriate model.

One possible method for supporting the hypothesis-based method is to use a library of models that describe particular architectural styles. For instance, suppose an analyst has a system that they believe uses a blackboard style. A model similar to the one shown in Figure 6 could be used to instantiate a model for the system in question.

Figure 6(a) depicts the knowledge source partition of a

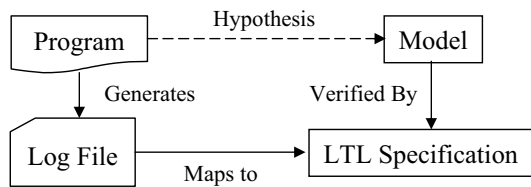
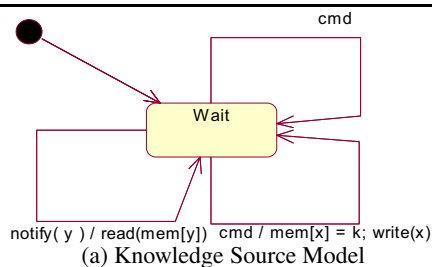
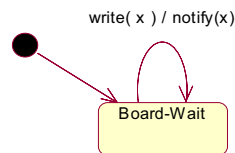


Figure 5. Hypothesis-based context

blackboard architecture. This model captures the fact that at various times a knowledge source executes a number of commands (represented as *cmd*). The two cases represent the fact that some commands result in a write event on the blackboard, resulting in a data change to some memory location *x*. In addition, the knowledge source may react to other events, including a write to some arbitrary memory location *y*. Figure 6(b) contains the blackboard partition of a blackboard model. In this model, the blackboard partition is responsible for handling notification when memory location *x* is updated.



(a) Knowledge Source Model



(b) Blackboard Model

Figure 6. Basic blackboard model

This basic blackboard model can be used to instantiate the components hypothesized to be part of a subject system. Specifically, if an analyst believes that there is a single blackboard with four knowledge source clients, the “model” of the system would be a statechart with four client components (with appropriate name modification) and a single blackboard component, as shown in Figure 7. An assumption of this architecture would be that embedded in each of the components is a finite-state model based on the one shown in Figure 6(b).

The challenge in using a hypothesized model lies in the mapping of events observed in a log file. For instance, in the basic blackboard model of Figure 6(b), the *write(x)* event must be mapped to events in a log file. The difficulty in this lies in identifying those log file elements that correspond to the *write(x)* event. Without the proper structural information necessary to determine number and nature of roles

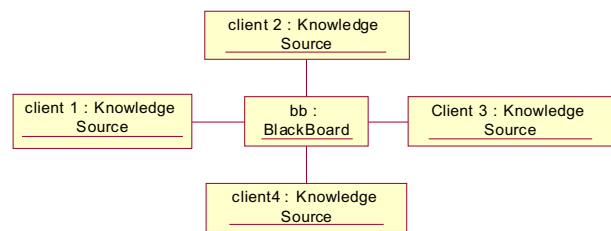


Figure 7. Blackboard architecture

played by components, the hypothesis strategy amounts to reconstruction in the dark.

4.3. Model abstraction from log files

Several approaches for performing event abstraction in order to derive state-based models have been suggested including work done by Cook and Du [10]. This form of reconstruction strategy requires the use of two sets of log files, where the first set is used to generate a candidate model and the second set is used to verify the model. Specifically, the first set of traces provides the data necessary to derive a candidate model and serves as input to any of the various event abstraction methods. The second set of traces, then, serves as an independent source of data with which to perform the verification task. With respect to such an event abstraction strategy, the verification framework would be focused on the second set of traces. Figure 8 depicts the verification framework in the context of an event abstraction approach. Specifically, rather than using code, one trace set serves as the source for model abstraction while the second trace set serves as the source for LTL mapping.

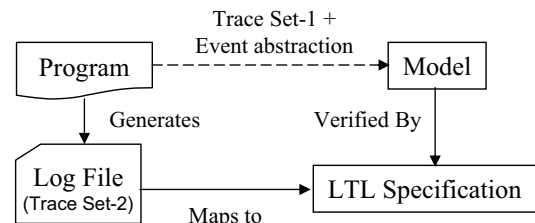


Figure 8. Event abstraction context

The advantage of event and model abstraction from a log file is that the representative techniques are systematic. With regards to the verification of these models, special care must be taken to generate two representative sets of traces to be used as the model abstraction source and verification test set. Specifically, it is ideal to ensure non-overlapping scenarios when generating the traces.

4.4. Stateless models from code

Any reverse engineering approach based on event abstraction has the disadvantage that a rich source of information, namely the source code, is being ignored. As a result, the potential for omitting important structural and behavioral partitions of a system are increased.

A large number of reverse engineering and design recovery approaches focus on the reconstruction of software architectures that are structural in nature. Such approaches rely on patterns and lexical hints via naming conventions to identify architectural styles and behavioral semantics.

One of the potential applications of the methodology being introduced in this paper is to use traces and log files as the means for verifying whether the behavior corresponding to a subject system adheres to a hypothesized style. In this context, a recovered structural model is used as a guide to determine partitioning. Figure 9 depicts this context, where the path from source code to model is achieved by the use of a stateless form of reverse engineering. The advantage of this strategy is that the introduction of the verification approach makes it possible to provide an extra level of assurance regarding the consistency of a reconstructed model with respect to dynamic information recovered from the execution of a subject system.

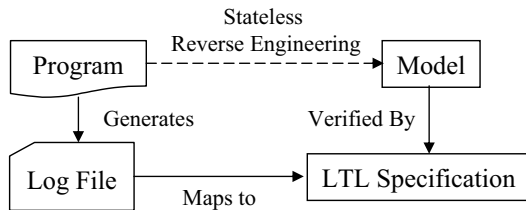


Figure 9. Stateless model context

As was the case with the hypothesis strategy, the stateless model strategy requires the use of some form of state model that is based on behavioral patterns for a given architectural style. On the surface, given this requirement, the stateless model appears to be no better than a hypothesis model. However, the stateless model strategy does have an advantage over a hypothesis-based strategy in that the existence of a recovered structural model provides information regarding the type and number of components that need to be instantiated with library state models.

4.5. Stateful models from code

The final strategy is the reconstruction of stateful models from source code. Models reconstructed using this particular strategy, while the easiest to verify, are the most difficult to derive. Figure 10 depicts the context for this strategy. The advantage of this strategy is that the models would require no modification outside of corrective changes resulting from failed verifications. Furthermore, since approaches that derive behavior and stateful models capture details embedded in code, this strategy increases the ability to verify more interesting properties such as safety and liveness rather than just those associated with a high-level semantics of a particular architectural style.

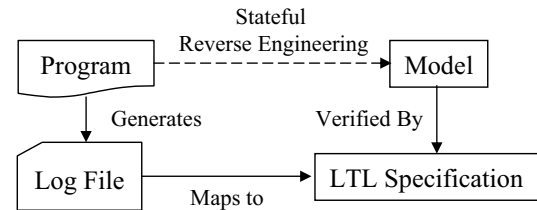


Figure 10. Stateful model context

5. Example

Figure 11 shows a model that was derived from the source code of a system that utilizes shared memory and semaphores in order to share resources through mutual exclusion (e.g., a dining philosophers implementation). In the model, the rectangles represent processes or “philosophers” while the square in the center of the diagram represents a shared memory repository. The smaller squares within the repository represent specific memory elements or locations, with the arrows indicating interest in specific locations.

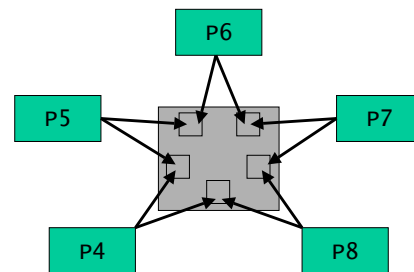


Figure 11. Derived structural architecture

From an architectural standpoint we are interested in determining whether the system in Figure 11 exhibits behaviors similar to that of a blackboard system. With respect to the strategies described in Section 4, the example in this section starts from a stateless model. The remainder of this section demonstrates steps in the verification process.

5.1. Log file generation

Figure 12 depicts a sequence of a log file generated from a program that implements the system given above. The event capture system that we have developed logs high-level information such as thread start and stop as well as method calls. The event capture system also has the capability to log object state changes that occur when attributes values are modified.

The potential uses for the log file shown in Figure 12 are two-fold. First, the log file can be used to identify whether the system meets certain architectural assumptions that may arise in during model construction (See Section 5.2). Second, the log file can be used to verify problem specific queries such as presence of mutual exclusion properties. With the latter option, derivation of a more detailed model

```

1.  Chopstick.putBack_chopsticks() -- ThreadID 296
2.  Philosopher.getSemaphore() -- ThreadID 297
3.  Philosopher.getLeft() -- ThreadID 296
4.  Semaphore.acquire() -- ThreadID 297
5.  Philosopher.getRight() -- ThreadID 296
6.  Philosopher.setState() -- ThreadID 296
7.  Philosopher.eat() -- ThreadID 298
8.  Chopstick.test() -- ThreadID 296
9.  Philosopher.getLeft() -- ThreadID 296
10. Philosopher.getRight() -- ThreadID 296
11. Philosopher.getState() -- ThreadID 296
12. Philosopher.getState() -- ThreadID 296
13. Chopstick.test() -- ThreadID 296
14. Philosopher.getLeft() -- ThreadID 296
15. Philosopher.getRight() -- ThreadID 296
16. Philosopher.getState() -- ThreadID 296
17. Philosopher.getState() -- ThreadID 296
18. Philosopher.getState() -- ThreadID 296
19. Philosopher.setState() -- ThreadID 296
20. Philosopher.getSemaphore() -- ThreadID 296
21. Semaphore.release() -- ThreadID 296
22. Chopstick.getChopstickObj() -- ThreadID 298
23. Chopstick.putBack_chopsticks() -- ThreadID 298
24. Philosopher.getLeft() -- ThreadID 298
25. Philosopher.getRight() -- ThreadID 298
26. Philosopher.setState() -- ThreadID 298
27. Chopstick.test() -- ThreadID 298
28. Philosopher.getLeft() -- ThreadID 298
29. Philosopher.getRight() -- ThreadID 298
30. Philosopher.getState() -- ThreadID 298
31. Philosopher.getState() -- ThreadID 298
32. Chopstick.test() -- ThreadID 298
33. Philosopher.think() -- ThreadID 296
34. Philosopher.getLeft() -- ThreadID 298
35. Philosopher.getRight() -- ThreadID 298
36. Philosopher.getState() -- ThreadID 298
37. Philosopher.eat() -- ThreadID 295

```

Figure 12. Log file of example

becomes necessary while the former option may be a general architecture.

5.2. State model construction

In the stateless model construction and verification strategy, an architectural structure is derived from code and an appropriate architectural style suggested. The architectural style choice often arises based on domain knowledge or other experiences gained from using a system. In the example system, a documented star topology could result in the assignment of a client-server system as a suggested style while the use of shared memory protected by semaphores further narrows down the choices to a repository, shared-memory, or blackboard architecture.

Figure 13 shows a portion of the finite-state model of the example system using the basic blackboard model from Figure 6(b). To complete the model, it is necessary to map model events where appropriate. Given that the basic models are abstract and potentially would only capture component interactions, the mapping of model events to log file events are likely to be injective but neither subjective nor bijective.

5.3. Verification

The objectives of the verification step are to determine whether the log file contains evidence that corroborates the information provided by a recovered architectural model.

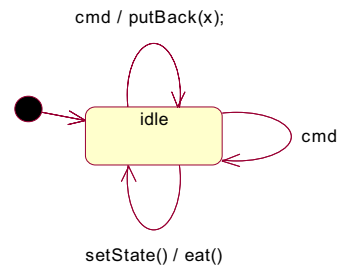


Figure 13. Instantiated state model

With respect to the example, the objective is to determine whether the given system follows a blackboard style of interaction between components. In order to perform this step, two activities are needed. First, a set of properties to be checked must be derived from a log file such as the one found in Figure 12. Second, the model must be verified using these property specifications.

To address the first activity, for the example we derived an LTL specification by first mapping the `putBack` event from line 1 to the `write` action of the basic model, the `setState` event from line 19 to the `notify` event of the basic model, and the `eat` event in line 37 to a `read` event in the basic model. The resulting LTL specification based on this mapping is as follows:

$$\square \neg \text{putBack} \vee \diamond (\text{putBack} \wedge (\neg \text{eat} \mathcal{U} (\text{setState} \vee \square \neg \text{eat}))) \quad (3)$$

which states that after `putback` event, notification of the release of the resource precedes the subsequent use of the resource. Using the Spin model checker we were able to verify that the property indeed exists and that furthermore, applies to all processes involved in the architecture. The property itself corresponds to the notification behavior of a blackboard system where an update to shared memory results in notification to the subscribers.

6. Related work

Mittermeir and Pozewaunig [5] developed a technique called *static behavior sampling* (SBS) which exploits the behavior of static components directly, without taking detour of externally added descriptions. The SBS technique calls on historical data about component executions, which is provided by test data. The data helps infer information about characteristic behavior of components, without having to analyze or execute the component. This approach lays the groundwork for ensuring that a generated log file has reasonable behavioral coverage.

Cook and Du [10] present a technique for discovering patterns of concurrent behavior from traces of system events. The systems event trace helps determine the concurrent behavior and acts as a major source for deriving a model. The technique is based on probabilistic and statistical analysis of the event traces. This approach is an example

of a strategy that would benefit from log file based verification.

Andrews [7] 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. The approach followed by Andrews uses log files and model checking to perform software testing. Our approach is analogous to the testing approach with an intent of verifying reverse engineered models.

Holzmann and Smith [11] developed an approach to software verification based on model extraction. The approach allows extraction of verification models mechanically from the source, which is then subjected to thorough verification using model checking techniques. They introduce a technique for extraction of verification models from the code, construction of test harnesses and testing these test harnesses against the models constructed. Holzmann therefore introduced a new software testing method, wherein the design instead of the code is tested against test harnesses using model checking. The approach presented in this paper also uses model checking, but to verify a reverse engineered model through the use of log files.

7. Conclusions and future investigations

Many approaches to reverse engineering have been suggested for the reconstruction of software architectures from source codes. The challenge of using these approaches lies not in the underlying technologies, but rather in the confidence that a user may or may not have in the end products. However, by combining the knowledge gained from using these technologies with knowledge obtained directly from an execution of a program, confidence can be increased using lightweight formal techniques.

Our experience with the approach has shown a great deal of promise with some moderate risks in each of the process steps. Currently the approach is being applied to a NASA funded project, that addresses timing and race condition verification of real-time systems. Execution traces that capture temporal behavior of real time software are analysed to uncover all possible derivations of program executions and race conditions. An ability to capture appropriate event information and map those events 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 Debugging Platform [8] in order to log events in Java programs without resorting to behavior modification as is possible with code instrumentation techniques. In addition, we are interested

in developing tools that allow analysts to browse a log file and map model events to log file events.

Finally, we are in the process of constructing a number of demonstrations of the approach on large scale systems using a variety of techniques that fall within all of the recovery classes described in Section 4 as a means of validating the framework.

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] S. Mancoridis, B. Mitchell, Y. Chen, and E. Gansner. Bunch: a clustering tool for the recovery and maintenance of software system structures. In *Proceedings of the 1999 International Conference on Software Maintenance*. IEEE, August 1999.
- [3] 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.
- [4] Kevin Templer and Clinton L. Jeffery. A Configurable Automatic Instrumentation Tool for ANSI C. In *Proc. of the Automated Software Engineering Conference*, 1998.
- [5] Roland T. Mittermeir and Heinz Pozewaunig. Self-descriptive Software Components. In *Proc. of the 16th European Meeting on Cybernetics and Systems Research (EMCSR 2002)*, Austria, April 2002.
- [6] Gerard Holzmann. The Spin Model Checker. *Transactions on Software Engineering*, 23(5):279–295, May 1997.
- [7] 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.
- [8] The Java Platform Debugger Architecture (jpda). [Online Available] <http://java.sun.com/products/jpda>.
- [9] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Patterns in Property Specifications for Finite-State Verification. In *Proceedings of the 21st Intl. Conf. on Software Engineering*, 1999.
- [10] Jonathan E. Cook and Zhidian Du. Discovering Thread Interactions in a Concurrent System. In *Proceedings of the 9th Working Conference on Reverse Engineering (WCRE 2002)*, pages 255–264, Richmond, Virginia, October 2002.
- [11] Gerard J. Holzmann and Margaret H. Smith. An Automated Verification Method for Distributed Systems Software Based on Model Extraction. *Transactions on Software Engineering*, 28(4):364–377, April 2002.