Zenet: Generating and Enforcing Real-Time Temporal Invariants

Chris Lewis
University of California, Santa Cruz
1156 High St, Santa Cruz, California, USA
cflewis@soe.ucsc.edu
`http://cflewis.com`

ABSTRACT

Generating correct specifications for real-time event-driven software systems is difficult and time-consuming. Even when such specifications have been created, they are often used to guide development rather than state properties guaranteed by the actual system. We propose a specification generator that reads execution traces and can generate invariants with real-time constraints. That specification can also offer programmers the ability to repair violated invariants at runtime. Creating fault-tolerant systems in this manner would provide software engineers guarantees about the software's high-level operation and its ability to recover from errors.

Categories and Subject Descriptors

D.2.1 [**Software Engineering**]: Requirements/Specifications; D.2.5 [**Software Engineering**]: Testing and Debugging

Keywords

specification generator, temporal invariants, runtime software-fault monitoring, rule engine, video games

1. PROBLEM STATEMENT

Over the last 20 years, researchers have been investigating automatically generating temporal specifications from software source code and execution traces. Specifications serve a variety of purposes, including code documentation, finding errors and validating test suites. Specification generators have commonly focused on deriving invariants that monitor control flow. While these low-level specification generators are useful, they are unable to reason about the functionality of the system under test (SUT) at a high-level, and their focus on source code means that the generated specifications are implementation dependent. Moreover, while these specifications are useful in and of themselves, their value would greatly increase if they were combined with fault-tolerance techniques that would allow the SUT to be repaired to within specification.

2. RESEARCH GOALS

We intend to generate sets of program invariants that form specifications. We believe creating these specifications from the event streams outputted by the execution of event-driven systems will create a powerful, scalable tool that can detect and enforce correct operation of a software system. Towards that end, we have the following goals:

- Utilize machine learning methods, such as association rules and temporal sequence finding, to learn invariants from event streams generated by software executions.
- Deploy these invariants in a runtime software-fault monitor built with a rule engine.
- Allow programmers to express the repair of invariant violations.

We will validate our research by creating a tool, "Zenet", which attempts to achieve these goals.

3. PROPOSED SOLUTION

We propose generating software specifications by monitoring *events* that occur in event-driven systems, and then giving programmers the opportunity to define repairs when the SUT deviates from the specification.

Events are an abstraction of input, output and state. Events, and the possible transitions between them, can be represented as an *event scene graph* (ESG) [2]. *Event sequences* are a specific walk through an ESG. Monitoring these event sequences allows us to view the operation of the SUT from a high, abstracted level, separated from the implementation and architecture of the software.

Our work is inspired by Yang & Evans' Perracotta who inferred temporal properties about software from event sequences [6], and Ernst et. al's Daikon [5] who derived invariants about a program's data structures. We intend to build upon these works by creating a system that combines elements of both systems: one that detects invariants with *real-time* temporal constraints, as well as finding various properties about the events themselves, such as maximum or minimum values. Our system differs by taking a data mining approach, utilizing large corpuses of traces that are created during software executions by end-users, generating invariants that reflect the software's actual, rather than intended, use.

As an example of a possible invariant, let us specify two events: a user clicking on a save button, and the file actually

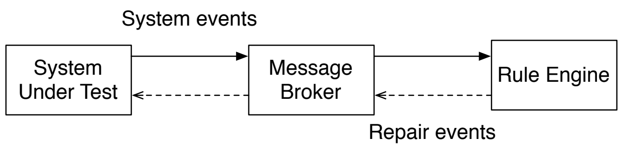

Figure 1: Architecture of Zenet's communication with the system under test. Events flow through the message broker, allowing the rule engine to run synchronously or asynchronously.

being written to disk. From this, we can then define an invariant (expressed formally in metric temporal logic [3]):

$$\Box(\mathit{SaveClicked} \rightarrow \Diamond_{<5}\mathit{FileWritten})$$

This can be read as, "For all states, when the save button is clicked, within five seconds, the file will be written."

If the file fails to save to disk after five seconds, the invariant is violated. Note that we do not take into account how many function calls are made or which software components are utilized; this abstraction describes the high-level operation of the SUT, as well as makes the specification implementation independent, able to function even after the SUT's source code has been refactored.

While a generated specification can show how a system functions, human-coded specifications are often used to verify that a system functions within certain limits as part of fault-tolerant computing techniques [1]. Such systems use specifications to apply repairs, and our specification can offer the programmer an opportunity to define a repair of a violated invariant, creating a fault-tolerant system. In our file saving example, attempting to save the file to a disaster-recovery folder could be a reasonable course of action. To detect when invariants have been violated, and how to apply these repairs, we turn to *runtime software-fault monitors.*

Runtime software-fault monitors compare the current state of a software system against a specification, creating a warning when a deviation from the specification is detected [4]. We intend to create a user-friendly, efficient monitor using a rule engine to monitor events fired from the SUT, with the rule engine analyzing the event stream against invariants. Once an invariant is violated, the rule engine can send an event back to the SUT that will enact a repair.

The events from the SUT, and the repair events from the monitor, will be mediated by a message broker, as illustrated in Figure 1. Brokers can use adapters to allow messages to be sent and received from a variety of languages, meaning the specification and its repairs are language independent.

4. RESEARCH APPROACH

We intend to answer questions about generating invariants from event streams, as well as the viability of rule engines as runtime software-fault monitors, by creating Zenet. By building the system ourselves, we hope to encounter and answer problems about:

- Approaches to instrumenting programs as streams of events.
- Which machine learning techniques are useful for analyzing event streams.
- Whether large numbers of execution traces lead to better results.
- How to identify meaningful invariants.
- How to employ a messaging system to allow the monitor to operate synchronously and asynchronously, as well as investigate performance differences between local or distributed operation.

Our current progress has been focused on preliminary studies using a rule engine as a monitor, with hand-coded specifications. We have instrumented a small Java video game called *Infinite Mario Bros.*, and are able to successfully monitor events in the game (such as jumping, landing and getting coins) and repair faults (using direct method calls).

5. INTENDED EVALUATION

To evaluate Zenet, we have decided to focus on solving problems within the video game domain. Video games are a difficult domain to debug, as developers intentionally create emergent behaviors in their virtual worlds that hinge on interactions between players and simulation systems. The state explosion that results from this emergence means that many bugs are resistant to automated bug exposure tools. We believe that if we are able generate specifications and specify repairs within this domain, then the proposed architecture will be robust enough to function in many other areas.

We will evaluate the quality of the generated specifications against the possible bugs that could be seeded into the game, as well as providing specifications to game designers to ascertain their readability and quality. We hope to deploy Zenet with a commercial product in order to verify its success under real conditions.

This process should rigorously test Zenet's capabilities, and provide a strong foundation with which to evaluate its contributions.

6. REFERENCES

[1] Avižienis, A., Laprie, J. C., Randell, B., and Landwehr, C. Basic concepts and taxonomy of dependable and secure computing. *IEEE TDSC 1*, 1 (January 2004), 11–33.

[2] Belli, F., Budnik, C. J., and White, L. Event-based modelling, analysis and testing of user interactions: approach and case study. *Software Testing, Verification and Reliability 16*, 1 (2006).

[3] Chang, E., Manna, Z., and Pnueli, A. Compositional verification of real-time systems. In *LICS '94 Proceedings* (July 1994), pp. 458–465.

[4] Delgado, N., Gates, A. Q., and Roach, S. A taxonomy and catalog of runtime software-fault monitoring tools. *IEEE TSE 30*, 12 (2004), 859–872.

[5] Ernst, M. D., Perkins, J. H., Guo, P. J., McCamant, S., Pacheco, C., Tschantz, M. S., and Xiao, C. The Daikon system for dynamic detection of likely invariants. *Sci. Comput. Program. 69*, 1-3 (2007), 35–45.

[6] Yang, J., Evans, D., Bhardwaj, D., Bhat, T., and Das, M. Perracotta: mining temporal API rules from imperfect traces. In *ICSE '06 Proceedings* (2006), ACM, pp. 282–291.