# Runtime Repair of Software Faults using Event-Driven Monitoring

Chris Lewis, Jim Whitehead
University of California, Santa Cruz
1156 High St, Santa Cruz, California, USA
{cflewis,ejw}@soe.ucsc.edu

## ABSTRACT

In software with emergent properties, despite the best efforts to remove faults before execution, there is a high likelihood that faults will occur during runtime. These faults can lead to unacceptable program behavior during execution, even leading to the program terminating unexpectedly. Using a distributed event-driven runtime software-fault monitor to repair faulty states creates an enforceable runtime specification. Using such an architecture would ensure that emergent systems operate within specification, increasing the reliability of such software.

## Categories and Subject Descriptors

D.2.1 [**Software Engineering**]: Requirements/Specifications; D.2.4 [**Software Engineering**]: Software/Program Verification; D.2.5 [**Software Engineering**]: Testing and Debugging

## Keywords

temporal invariants, runtime software-fault monitoring, event-driven systems, specifications, message broker, rule engine, video games

## 1. INTRODUCTION

The scope and complexity of modern software development has led to the development of many processes to verify the quality of software. Techniques such as model checking, static code analysis and unit testing are all employed for this task. However, much of today's software has such complex state possibilities that such techniques cannot explore and exercise the entire state tree. This is particularly pronounced in software that has emergent behaviors, creating a complex system from simpler interactions, where user exploration of the interaction space is a key focus.

In software with emergent properties, despite the best efforts to remove faults before execution, there is a high likelihood that faults will occur during runtime. We propose the detection and repair of these faults in systems by utilizing an easy-to-use declarative rule engine as part of a runtime software-fault monitor, analyzing events emitted from the system under test (SUT).


*ICSE* 2010 Cape Town, South Africa


In contrast to many other monitors that only observe execution and log errors, we allow programmers to specify a desired repair when an invariant violation in the SUT is detected. This dynamic enforcement of a documented, human-readable specification creates a powerful, reliable, fault-tolerant system.

By monitoring events, rather than logical transitions, we have developed an architecture that is both language-independent and machine-independent from the SUT. This architecture is being developed as part of a larger tool called "Zenet."

This demonstration shows the viability of the technique using human-authored rules. We apply the architecture to a Java implementation of *Super Mario World* creating a program we call *Lakitu*. We show how bugs in the code can lead to faults, and how the action taken by the rule engine enforces the game's integrity.

## 2. RELATED WORK

Delgado et al. [4] provides an excellent summary of influential monitors and offers a common language with which to describe and categorize them. Two of the monitors covered by Delgado et al. have strong correlations to our approach: MaC and DB-Rover.

The MaC system is a project at the University of Pennsylvania that allows a user to specify requirements of a target program, which MaC will then appropriately instrument and verify whether the execution history meets the original specification [8]. Of particular interest is RT-MEDL, an extension of their event definition language that allows for real-time constraints to be placed on event rules [14]. Our architecture also allows for the specification of real-time constraints, but instead of compiling specifications into a formal logical language, we employ a user-friendly syntax with our rule engine which is then parsed into a Rete tree [5].

DBRover, like MaC, is a monitor that uses a temporal logic with real-time constraints to verify runtime correctness. However, DBRover encourages a stronger decoupling of the SUT and the verification process than MaC. It operates on a separate validation server, monitoring a database for execution integrity, and can capture data for analysis at a later date. Inspired by this decoupling, we use a message broker to pass events to our runtime monitor, providing a language-independent and machine-independent communi-


**Figure 1: A screenshot from Lakitu. The menu bar allows the user to enable or disable the rule engine, and switch the game version between the correct code and the buggy code.**

cation mechanism.

The value of repairing software at runtime with manually-specified repairs has been previously illustrated with the success of the error detection and handling created as part of the IBM MVS/XA operating system [10]. By manually inspecting and repairing inconsistent data structures, the reliability of the system was increased by an order of magnitude [6].

More recently, there has been research that combines specifications with runtime repairs, of which the most notable is ClearView [11]. ClearView is able to infer the invariants of a program by monitoring x86 instructions from normal executions. If ClearView detects an invariant violation, it halts the program and begins to generate candidate patches to repair the binary. Once a repair has been found, the program is restarted. Unlike ClearView, we apply repairs while the program is still executing, much like the MVS/XA system, and in-line with Rinard's Acceptability-Oriented Computing approach [13]. In contrast with ClearView's generated specifications, Lakitu operates at a much higher-level, using events to describe the high-level operation of the SUT. This means the specification communicates a SUT's operation, rather than a specific implementation.

Lakitu can be seen as a combination of the real-time temporal specifications provided by RT-MaC, the decoupling ethos of DBRover, and the repairs of ClearView. We combine these technologies with a rule-based specification to verify and enforce the correct high-level operation of the SUT. This novel approach creates a powerful, yet user-friendly, architecture.

## 3. LAKITU

Lakitu is a clone of *Super Mario World* built on an open-source Java project called *Infinite Mario Bros.* [12]. *Super Mario World* is a game where a character called Mario moves through a 2D level. He can jump up onto platforms and across gaps, collecting coins and jumping onto enemies. A screenshot of Lakitu can be seen in Figure 1.

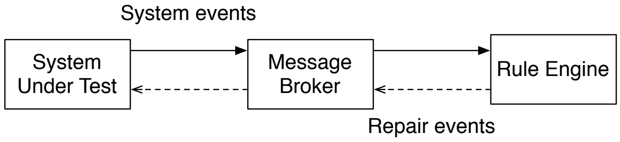


**Figure 2: Architecture of our tool's communication with the system under test. Events flow through the message broker, allowing the rule engine to run synchronously or asynchronously.**

Video games are an excellent domain to investigate monitoring and repairing event-based systems. Their core functionality hinges on providing emergent systems that offer players choice. Many players will choose to perform actions that the programmers never intended, which can lead to undesirable states. Video games also offer tight technical and temporal restrictions, rigorously exercising the efficiency of any architecture. We believe that if our architecture is successful in the video game domain, it will be robust enough to be applied to many others.

Lakitu has two modes of operation: one where the game is "correct" (ie. the code is unchanged from *Infinite Mario Bros.*) and one where the game is "buggy," where we intentionally introduce bugs in the code base. Examples of bugs include Mario jumping too high, too many enemies being spawned and Mario receiving two points for every coin, rather than one. The rule engine can be enabled or disabled, and users can see the results of the rule engine's involvement on the buggy code. A console window can be displayed, showing textual logs that allow users to see the internal operation of Lakitu.

To decide which bugs to introduce into Lakitu, we have been working on a taxonomy of video game bugs (not yet published). While a lengthy discussion of the taxonomy is outside the scope of this paper, it is divided into two main groups, "implementation errors" and "design errors." Lakitu focuses on implementation errors. Implementation errors are split between bugs that have no temporal aspect (such as an object being in an invalid position), and bugs that can only be expressed as a temporal issue (such as an object being in a valid position, but moving incorrectly over time). Where applicable, we chose to introduce a bug from each category of the taxonomy. For the position categories, we allowed Mario to jump too high, and also allow Mario to hover in the air at a height that is otherwise permissible. We will revisit these specific bugs later in the paper.

## 4. TECHNICAL FOUNDATIONS

### 4.1 Overview

Our tool monitors events emitted from *Infinite Mario Bros.* Events are an abstraction of input, output and state. In our example, such events include Mario jumping, landing, or collecting a coin. Events, and the possible transitions between them, can be represented as an *event scene graph* (ESG) [3]. *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 high-level architecture can be seen in Figure 2. The instrumented SUT sends event messages to a message broker, which in turn directs messages to the rule engine. These events are added to the rule engine's knowledge base, and then evaluated against a human-authored specification to see if any rules are matched. Matching a rule indicates that there has been an invariant violation. With a fault detected, the programmer can choose a repair action to take, specified as an event to be enacted, which is then sent to the SUT via the message broker.

## 4.2 Event Instrumentation

The first step in our workflow is to instrument the SUT to emit important events to the message broker. We use events so that we can see the important aspects of the SUT's operation. In Lakitu, those events could be Mario jumping or landing, in an email program they could be starting a network connection or receiving an email, and in a music player they could be starting a song or adding album artwork. These events characterize the operation of the software, not the implementation.

We use events so that the specifications created by the human-authored rules communicate useful, high-level documentation about the SUT's operation. In many scenarios, non-programmer knowledge experts, such as game designers, dictate the specification of the program. However, they have no way of knowing how well the program's implementation matches their original intent. Using high-level events allows the specification to be handed to non-programmer knowledge experts to verify its correctness.

Additionally, instrumenting events in such a way makes the specification resistant to refactoring of the SUT's codebase, allowing the rule file to be used as an aid to the development process, rather than a verification of correctness at the end of the development cycle.

The drawback of monitoring high-level events is that they must be instrumented manually, as there is no way of knowing which parts of a computer program represent the "important" aspects. Other approaches, such as that taken by MaC, instrument function calls directly. This methodology is common, and is proven to work successfully, but it is too low-level for our purposes.

## 4.3 Message Broker

Emitted events are sent to a message broker. In our current implementation, messages are sent via the Java Messaging Service to an instance of ActiveMQ [1] that runs on the same machine as the Lakitu client. The broker can then send on events to a consuming program to be processed and delivered to a rule engine. Neither the broker, nor the rule engine, needs to operate on the same client machine as the SUT. This allows for processing to occur across multiple networked machines.

ActiveMQ offers language adapters for a number of common languages such as Java, C and Ruby. By separating the communication channel to a message queue, we achieve language-independence, allowing us to process messages in the Java-based Drools rule engine from a SUT written in any of the ActiveMQ supported languages.

Research has shown that ActiveMQ can handle over 30,000 messages being passed every second to a single consuming program, far greater than Lakitu requires [7].

In Lakitu, we send events asynchronously to the broker, and check if there are any waiting messages from the broker every frame. One can easily change this policy to send a message and synchronously wait for a reply (which could be a no action message) from the rule engine before continuing execution.

```
1  rule "marioJumpTooLong"
2    duration(2s)
3    when
4      $jump : Jump()
5      not(Landing(this after[0s,2s] $jump))
6    then
7      System.err.println("Mario jumped too long");
8      broker.send(session.createObjectMessage(new
           MarioMovement(...)));
9  end
```

**Figure 3: A rule that detects if Mario has been in the air for too long.**

## 4.4 Rule Engine

Once event messages are received, they are passed to the rule engine. As with the message broker, the rule engine runs on the Lakitu client machine. The rule engine fires after every event is inserted into the working memory. Lakitu uses an open-source Java engine called Drools [2]. We chose Drools for its declarative, easily-readable syntax and its support for Complex Event Processing (CEP) [9]. CEP allows us to easily write rules that are dependent on real-time constraints. Programmers write rules as characterizations of faulty states or event sequences. These are a specification of invariant violations. In emergent systems, the number of correct states is typically larger than the number of faulty ones, so it is more concise to encode invariant violations rather than the invariants that indicate correctness.

An example rule from our rules file can be seen in Figure 3. The first line begins the definition of the rule. The second line provides a duration on the rule, telling Drools that this rule should be checked to be true in two seconds; this allows time for a landing event to come in after the rule is activated when a jump event is found. Lines 3 – 5 specify the condition on which the rule will be successfully matched. The condition specifies what characterizes the fault we wish to detect. Here, we search the working memory for a jump event. If one is found, the rule engine waits to see if there isn't a landing event timestamped within two seconds of the jump event. Thus, this rule detects if Mario doesn't come down within two seconds, meaning he has been in flight for too long. Line 8 sends a message back to the broker indicating that Mario's movement should be altered, moving him towards the ground (the details of the implementation has been excluded for brevity).

A common concern of this approach is that bugs are just as likely to appear in the rule file as they are in the implementation itself. Our experience hints that this is not the case. The rule file is orders of magnitude more compact than the implementation, making any logic bugs far shallower than the original system. This is combined with an easily-readable, yet powerful syntax that can express complicated ideas such as temporal limitations. This means non-programmer knowledge experts can verify the rule file to be correct. Given these benefits, we believe the rule file, while not impervious to error, has a far greater chance of correctly expressing the software specification than the original code base.

### 4.5 Repairs

Repairs are sent back to the game as *repair events*, which are events that should be enacted, rather than emitted. In Figure 3, we see a `MarioMovement` event, which affects the Mario object's acceleration. Other events request the translation, creation or deletion of objects; as well as the modification to internal counters such as the score.

These repairs do not change the state of objects directly, but rather express an *intent* that the SUT must handle. This removes the need for large amounts of implementation-dependent logic in the rule engine, as well as providing the SUT an opportunity to appropriately schedule the fix, or even ignore it if need be.

In Lakitu, events are sent to the rule engine asynchronously every frame. This most accurately represents the deployment of such an architecture in a commercial video game; games are too computationally expensive to afford any processing time to wait on a network connection. However, continuing processing means that some bugs that are particularly time-dependent can create visible faults to the user. An example of such a bug could be specifying Mario's jump height to be too high: the acceleration required to reach that height within the correct time moves Mario fast enough to be a user-visible fault. Lakitu runs at 24 frames per second, and we typically observe a lag of one to two frames of a fault occurring before a repair is enacted. In contrast, if we run Lakitu synchronously, the bug can be repaired within one frame, but runs the risk of the rule engine response time stalling the drawing of the game. Lakitu is designed to be reactive, repairing faults after they occur, but working in a synchronous manner opens the possibility of ascertaining whether the state is correct before drawing to screen, and changing the incorrect behavior before it is ever enacted.

In other software domains, where the small wait required for a response from the rule engine is acceptable, designing repairs for synchronous operation would be preferable, preventing failures being made apparent to the user.

## 5. CONCLUSION

In this paper, we have presented Lakitu, a video game demonstrating the benefits of repairing software at runtime. We employ an event-driven architecture, that is both language-independent and machine-independent, which allows programmers to easily specify human-readable specifications that are enforced when the program is executed. This ensures the reliability of emergent systems at runtime.

We believe employing such a verification system dramatically increases the reliability of the emergent properties of complex software, and will allow programmers to experiment with even more expansive, inventive and creative systems, assured that the system will operate within requirements.

## 6. REFERENCES


[1] Apache ActiveMQ. Available from World Wide Web: http://activemq.apache.org/.

[2] Drools. Available from World Wide Web: http://www.jboss.org/drools/.

[3] 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).

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

[5] Forgy, C. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artifical Intelligence 1* (1982), 17–37.

[6] Gray, J., and Reuter, A. *Transaction Processing: Concepts and Techniques (The Morgan Kaufmann Series in Data Management Systems)*, 1st ed. Morgan Kaufmann, September 1992.

[7] Henjes, R., Schlosser, D., Menth, M., and Himmler, V. Throughput performance of the ActiveMQ JMS Server. In *Kommunikation in Verteilten Systemen (KiVS)*, Informatik aktuell. 2007, ch. 10, pp. 113–124.

[8] Kim, M., Viswanathan, M., Kannan, S., Lee, I., and Sokolsky, O. Java-MaC: A run-time assurance approach for Java programs. *Formal Methods in System Design 24*, 2 (March 2004), 129–155.

[9] Luckham, D. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems.* Addison-Wesley Professional, May 2002.

[10] Mourad, S., and Andrews, D. On the reliability of the IBM MVS/XA operating system. *Software Engineering, IEEE Transactions on SE-13*, 10 (September 2006), 1135–1139.

[11] Perkins, J. H., Kim, S., Larsen, S., Amarasinghe, S., Bachrach, J., Carbin, M., Pacheco, C., Sherwood, F., Sidiroglou, S., Sullivan, G., Wong, W. F., Zibin, Y., Ernst, M. D., and Rinard, M. Automatically patching errors in deployed software. In *SOSP '09: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles* (New York, NY, USA, 2009), ACM, pp. 87–102.

[12] Persson, M. Infinite Mario Bros. Available from World Wide Web: http://www.mojang.com/notch/mario/.

[13] Rinard, M. Acceptability-oriented computing. In *OOPSLA '03: Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications* (New York, NY, USA, 2003), ACM, pp. 221–239.

[14] Sammapun, U., Lee, I., and Sokolsky, O. RT-MaC: runtime monitoring and checking of quantitative and probabilistic properties. pp. 147–153.

# 7. DEMONSTRATION

## 7.1 Setup

Lakitu will be set up to run on a laptop, with one window showing the game, and one showing the output from the rule engine. Session attendees will be able to play through the game, turning on and off the rule engine and switching between the correct and the buggy code.

The game can be restarted at will, setting the game back to Level 1 and clearing the rule engine's knowledge base.

We will stand next to the demonstration and help users to start the game, discuss with interested parties how Lakitu works, and bring a poster for others to read.

## 7.2 Interaction with Lakitu

Figures 5 – 10 show the interaction options for Lakitu. We would recommend the following play-through to interested parties:

- Load the game, with a console window running side-by-side, so players can see the event passing and rule engine evaluation in operation. Figure 4 shows an example of this output.
- Start the game with the good code, and the rule engine on.
- Begin playing a level, noticing the events being created through play.
- Turn off the rule engine, and change the code to the buggy code.
- Continue playing, noticing flaws in the game.
- Enable the rule engine, and evaluate how the rule engine sends repair events back. Note the similarities to the correct code again.

We anticipate that players will be able to see the benefits of Lakitu in around two minutes, with a five minute play session fully satisfying curiosity.

```
MarioComponent 15:02:57 Sending fact: edu.ucsc.eis
     .mario.events.Jump@1d402894
MarioComponent 15:02:57 Inserting event: edu.ucsc.
     eis.mario.events.Jump@5b7fd935
Rule_marioJumpTooLong_0 15:02:59 Mario jumped too
     long
MarioComponent 15:02:59 Handling repair: edu.ucsc.
     eis.mario.repairs.MarioMovement@f78d7c3
```

**Figure 4: Sample output showing a jump failure (no landing is detected within two seconds). The jump fact is sent from the SUT to the rule engine, that then inserts it. The rule engine doesn't get a landing, so fires a repair event which is handled back at the SUT.**


**Figure 5: The title screen for Lakitu. This is the first screen users see when they start the game.**


**Figure 6: This screenshot shows the version menu. This allows users to switch between the good code and the buggy code. This change can also happen dynamically in-game.**


Figure 7: This screenshot shows the rule engine menu. This allows users to enable or disable the rule engine. This change can also happen dynamically in-game. When the buggy code is running, enabling or disabling the rule engine illustrates how the repairs effect the game.

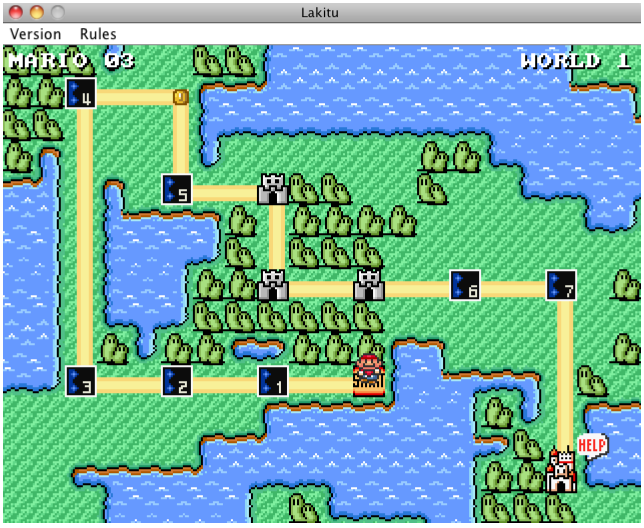


Figure 8: This is the map screen from where players can move to the next level, which is dynamically generated by the *Infinite Mario Bros.* codebase. If the buggy code is enabled, if a pit is generated in the new level, it is generated to be impossible to jump across. If the rule engine is also enabled, it requests the gap to be filled in with blocks to enable the player to jump across.


Figure 9: A screenshot from Lakitu in-game. This is a screenshot of Lakitu in normal operation.

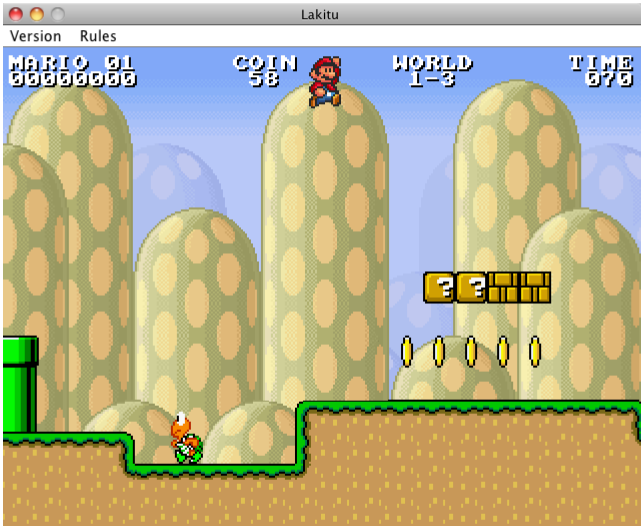


Figure 10: A screenshot from Lakitu in-game. This is a screenshot of Lakitu in buggy operation, with the rule engine disabled. Notice how high Mario has jumped: a jump of such height should not be possible. Re-enabling the rule engine will prevent Mario jumping so high again.