Repairing Games at Runtime or, How We Learned to Stop Worrying and Love Emergence

Chris Lewis and Jim Whitehead, University of California, Santa Cruz

// *By repairing games as they execute, Mayet lets developers focus on creating excellent gaming experiences without worrying about edge cases and untraceable bugs.* //

CAST YOUR MIND back to some of your greatest gaming moments (or ask your gaming friends for theirs). Maybe they include jumping a quad bike into an airplane (http://bit.ly/gtajump). Perhaps it was the simple joy of being killed in an online game by a traffic cone (http://bit.ly/halo3cone). Or maybe it was launching yourself into the air by riding an exploding gas canister, then hijacking a helicopter on the way back down (http://bit.ly/justcausejump). However, for all the good times, there are also the bad: maybe the time you fell through the floor (http://bit.ly/crysissink) or when your AI buddy's pathfinding sent him in the opposite direction (http://bit.ly/falloutbugs). How about when you couldn't do any damage at all to opponents in an online match (http://bit.ly/gtaonlinebug)? (Note: some of the games mentioned and referenced in this article feature violent content in the context of first-person shooter gameplay. *IEEE Software* magazine is not responsible for content placed on unaffiliated websites.)

These situations are created by *emergence*, the result of multiple simple interactions operating together to create a complex set of behaviors.[1] According to Katie Salen and Eric Zimmerman, emergence is the key requirement for games to remain continually engaging, as players will return to a game as long as it provides "variety, novelty, and surprise."[2] Think of Tic-Tac-Toe. For a six-year-old, Tic-Tac-Toe is an exciting game of strategy and intrigue. Who will win when two great, young minds collide? Of course, as we grow older, we realize that playing Tic-Tac-Toe with any reasonable competitor will result in a draw. The game can no longer surprise, and we abandon it for more varied pursuits, such as chess or Go. In software engineering, emergence is often considered an undesirable confluence of complex components, indicating an overengineered or poorly understood system, but in video games, emergence is king.

Emergence presents a significant verification obstacle: If software cannot be predicted, how can it be verified? Most automated testing techniques only function when the developer knows both the possible inputs and their corresponding outputs. However, those outputs are, by definition, unknowable (if we did know them for all possibilities, we wouldn't have an emergent system). Furthermore, even if we could begin the onerous task of enumerating possible inputs and outputs, emergence can only be achieved for an adult mind through state space explosion, and our ability to achieve test coverage across those states decays exponentially. Current automated techniques don't work, and the video game industry has resigned itself to hiring human testers,

0740-7459/11/$26.00 © 2011 IEEE

sometimes numbering in the hundreds, to manually test games.[3]

Complete verification of emergent software is not yet possible. With this in mind, we realized that we had to turn the problem of testing games on its head: If we can't know or enumerate the extent of all possible outputs at test time, what if we instead use the actual output of the system at runtime? With that output, we can then repair the system if it displays emergent misbehavior, creating a system that can recover from failures.[4,5] To do this, we created Mayet, a system for monitoring software at runtime. Mayet evaluates the running software's state for invalid states. If it detects an erroneous state, it instructs the software to repair itself.

Failures can occur in complex systems even if there are no faults in the individual software components.

The Basics of Bugs

When we talk about "bugs," what do we actually mean? Algirdas Avižienis and his colleagues define bugs at different points.[5] A *fault* causes an error. Faults can be caused by several things, such as human mistakes when coding or vulnerabilities that let external faults harm the system. An *error* is the system state that might lead to a failure. Some errors are handled and corrected in the system (such as packet loss in the TCP/IP protocol), but some can reach the system's external state, in which case a failure occurs. A failure occurs when errors cause a deviation from the system's expected behavior.

Failures can occur in complex systems even if there are no faults in the individual software components (in many domains, integration testing will expose these problems). Additionally, in non-safety-critical software such as games, failures aren't necessarily a problem if they don't cause significant harm. For example, if a nonplayer character (NPC) isn't spawned in a game world, this is indeed a deviation from the expected state and a failure has occurred. However, if the NPC has no critical interactions in the game, the failure's impact is negligible. In this article, we talk about acceptable failures, when a failure has occurred but gameplay isn't significantly impacted, and unacceptable failures, where the failure causes a strongly negative experience.

We adopt the term "system under test," which is used often in software verification. SUT refers to whatever system we're testing; in this article it will be a Mario-style game. Using SUT avoids confusion when describing Mayet, as often words like "system" and "program" can get conflated together and no one is really sure what they refer to anymore (authors included).

Now armed with our knowledge of bug definitions, we can investigate an actual problem.

Diving In: The Crysis Problem

Consider a single instance of the problems caused by emergence. While browsing YouTube for video game failures (see the sidebar), one of particular interest caught our eye. The video is of a playthrough of Crysis, a popular first-person shooter game where the player takes the role of protagonist, shooting enemies while trying to reach the end of the level. The player is on the runway of an aircraft carrier that has come under alien attack, with explosions going off all around. A failure causes the player to glitch through the floor to the level below, which is bad enough. Worse still, as the player attempts to go back to the starting point, he or she finds that a locker has fallen into a doorway. From the video, it's unclear why the failure occurred. Regardless, the player can't return to the top of the aircraft carrier, and must reload a saved game.

The astute reader might well comment that the failure was the player falling through the floor, and the locker issue is simply part of a failure cascade. Although this assessment is indeed correct, it provides cold comfort for our poor player, who must deal with the situation presented, regardless of how it occurred. The player's inability to move through a necessary doorway presents an unacceptable failure, in which the game has essentially reached an end state from which the player can no longer progress. Any number of situations could lead to a particular failure. In this case, the physics engine might have created an explosion that moved the locker to that position, the designer might have placed the locker to prevent backtracking, a bug in the implementation might have spawned the locker by accident, the locker might have been incorrectly flagged as immovable—the list goes on. Instead of looking at causes, we focus on the problem: the player can't progress because the critical path is blocked. Two simple solutions present themselves: remove the locker from the game world, or teleport the player back to where we expect him or her to be. Detailing the problem and possible repairs is far easier in this situation than trying to find the causes. In fact, we hypothesize that most problems in emergent software have many more possible causes than solutions.

The Crysis problem shows us two things:

- Problems can arise from component orchestration, not just components themselves. Jonathan Blow describes this as the *cross-cutting concerns problem*, noting that bringing algorithms into a "harmonious whole" is fraught with difficulty and no existing programming paradigms address this problem.[6]
- Although trying to find the cause of a problem is difficult, identifying the problem and repairing it is comparatively easy.

We designed Mayet with these insights in mind.

The Mayet Architecture

Mayet can be characterized as a runtime monitor. Runtime monitors watch the SUT as it executes via some means of instrumentation and can take some action based upon the current execution. Systems such as Rainbow[7] and ClearView[8] have begun to repair executions that are exhibiting some undesirable behavior. Mayet takes such an approach.

The SUT is manually instrumented to produce events across a message queue. Events are small messages indicating something important has happened and are parameterized with useful values. For example, a 2D platform game (such as Super Mario World) would have events for jumping, landing, and changing the score (parameterized with the old and the new score). The SUT produces event messages and sends them to a message broker, which routes them to a consuming rule engine running as a separate process. The rule engine evaluates the messages and checks whether they violate human-authored integrity rules. If an integrity rule is violated, the rule engine sends a message back to the SUT to perform a repair. An experienced programmer writes the rules beforehand to identify possible failure scenarios and how they can be repaired. The experience required to predict what failures could occur is roughly analogous to that required to create good unit tests that find likely problems at a function level. These rules specify safety properties, which express that something bad doesn't happen (such as getting stuck in a wall). We can't specify liveness properties, which are properties that say something good must eventually happen. Take the example that a player must eventually pick up a key. For that "eventually" to be truly evaluated, we would need to be able to verify an infinite trace. Model checkers can do this, but our dynamic evaluation can't. If we bound the property, such as a player must pick up a key within 30 minutes, we actually have a safety property.[9] In practice, most properties we want to specify are bounded. For example, knowing that a player, given infinite time, will eventually pick up a key doesn't ensure an enjoyable user experience.

The execution of the SUT and the rule engine are deliberately kept separate. There are several good reasons for this.

The first reason is *language independence*. Most games are written in unmanaged languages like C or C++. Rule engine implementations, such as Drools (www.jboss.org/drools) and Windows Workflow (http://bit.ly/windowsworkflow), are typically written in managed code, such as Java or C#. Separation lets us use whichever language we choose for the game, without needing to write a new rule engine each time.

A second reason is *implementation independence*. Because the SUT and rule engine communicate via messages, any refactoring of the SUT won't require altering the rules, provided the SUT conforms to the same specification.

Finally, SUT and rule engine execution are kept separate to ensure *execution independence*. Although rule engines are fast, they aren't without performance cost. Communication via messaging means the rule engine can be run asynchronously in a separate thread, CPU core, or even a completely different machine, allowing the SUT to run at full speed. Asynchronous execution trades off, with failures being possibly visible for a few frames before they're repaired. These failures are acceptable because they only appear briefly. If so desired, synchronous execution could be enabled, meaning the system could fix easily recognizable failures before they're rendered to the screen.

A Demonstration: Lakitu

Lakitu is a version of Super Mario World built on top of the Infinite Mario Bros open source 2D platform engine. We altered the engine by adding faults, instrumenting the game, attaching Mayet, and enacting repairs.

Adding Faults

We added faults by looking at our video game failure taxonomy (see the sidebar) and inserting various faults covering different categories. The bugs we inserted include accelerating Mario too much when he jumps (invalid position), having coins equal two points instead of one (invalid value change), and

> Events are small messages indicating that something important has happened.

GAME FAILURE TAXONOMY

To know how well a system-recovery tool performs, we need a metric of the domain's failure types. To help us evaluate Mayet—and future systems that attempt to aid game stability—we created a taxonomy of game failures.[1] We divide the categories into nontemporal (that is, they can be evaluated by a single inspection of the state) or temporal (that is, multiple inspections are required to ascertain their presence). We created these categories independently of Mayet's development.

Nontemporal categories include the following:

- *Invalid position*: an object is in a place it shouldn't be.
- *Invalid graphical representation*: an object doesn't appear correctly, or doesn't appear at all.*
- *Invalid value change:* an internal counter (such as score and health points) has been modified in a way that isn't allowed (for example, score going up too quickly or health points not going down).
- *Artificial stupidity*: the AI has done something that doesn't meet expectations, such as running over a land mine or standing in a doorway.
- *Bad information*: players get information about what they shouldn't do, such as seeing through walls, don't receive necessary information, or receive information out of order (such as being asked to kill a character the player has already defeated).
- *Bad actions*: a character takes an action when that action is disallowed or an action that must be taken isn't possible.

Temporal categories include the following:

- *Invalid position over time* categorizes invalid movements (for example, an object moving too quickly or too slowly, hovering when this isn't possible).
- *Invalid context state over time* observes objects as finite state machines; an object is in incorrect state for a wrong period of time (such as being stunned for too long).
- *Invalid event occurrence over time* is when an event happens too frequently or not frequently enough (events can be long sequences, not atomic events as with Mayet).
- *Interrupted event* occurs when an event that previously happened has stopped before reaching completion (events can be long sequences, not atomic events as with Mayet).
- *Implementation response* is when the game doesn't provide feedback quickly enough (for example, problems cause network or controller lag).*

Categories marked with an "*" are those that Mayet can't yet cover.

Reference

1. C. Lewis, J. Whitehead, and N. Wardrip-Fruin, "What Went Wrong: A Taxonomy of Video Game Bugs," *Proc. 5th Int'l Conf. Foundations of Digital Games* (FDG 2010), ACM Press, 2010, pp. 108–115.

spawning Bullet Bills too frequently (invalid event occurrence over time). Our insertions took the form of `if` statements that checked whether a flag had been set for buggy operation, which would then modify various data structures in Lakitu to have invalid values.

Instrumenting the Game

Instrumenting Lakitu manually sounds like it would be an alien and laborious task, wading through source code trying to find various points of importance, packaging a message with various values, and then sending it on the message queue for the rule engine to evaluate. In fact, the process is quite similar to that all-too-familiar programming task of logging.

When we instrument a game, our intent is to expose details about important events that occur during the game's execution. These are roughly analogous to the same details and events that programmers would place in a debugging-level log file for offline analysis. Programmers are already well-trained in finding these areas of execution and identifying which values should be emitted. In Lakitu, instead of writing a call to a logging framework, the call is a message to be sent. To do this, the game does four things:

- starts an ActiveMQ instance to act as a message broker (optional),
- creates a producer queue on the ActiveMQ instance that Lakitu sends messages out on,
- Mayet starts to consume the messages and send repairs back, and
- Mayet sends events as messages across the producer queue.

A *message broker* manages the routing of messages to different subscribers depending on various rules. We don't need a message broker in this case because we only run one instance of Mayet, so we could just route messages directly to a socket. However, if we required more complexity, such as redundant Mayet instances, or queuing messages if no Mayet instance is found, the message broker would become useful. In the actual Lakitu source, the producer and consumer queues are shared between Lakitu and Mayet for simplicity. In production, different queues should be used.

To create the messages, we serialize event objects, which are immutable objects of various values. A jump event, for example, records Mario's x and y positions, as well as his acceleration. Some events don't require values, such as when Mario dies (although the player might want to modify this to know exactly how he died).

When developing Lakitu, we found that our instrumentation method was largely iterative. After adding the bugs, we began to think about the information the rule engine would need to detect them. In the real world, of course, game programmers would use their experience to identify possible failure scenarios and add the events required to detect them. Once we knew what was required, we could create a new class (or modify an old one) to store those values and then find a suitable place for the event to be emitted.

Lakitu uses a standard run loop as seen in many simple games: input is read, the logical state is modified, and the output is rendered to the screen. When a game object modifies its state, Lakitu generally emits an event message. These messages weren't difficult or invasive to insert. More complex games might have more difficult state representations that are across separate threads and components. As long as these components can send messages to the queue, this shouldn't pose a concern.

Creating event messages and finding the correct places to insert them is a fairly simple and fast process. However, some thought might be required about how best to represent something. For instance, we needed a way of knowing whether Mario had fallen into a pit. The pit itself isn't an object, so no function is called when Mario falls down. Our buggy implementation disabled an if statement that kills Mario when he falls off the screen. We decided to send out Mario's location to Mayet once per second, so Mayet could also perform a boundary check and kill Mario if he falls off screen.

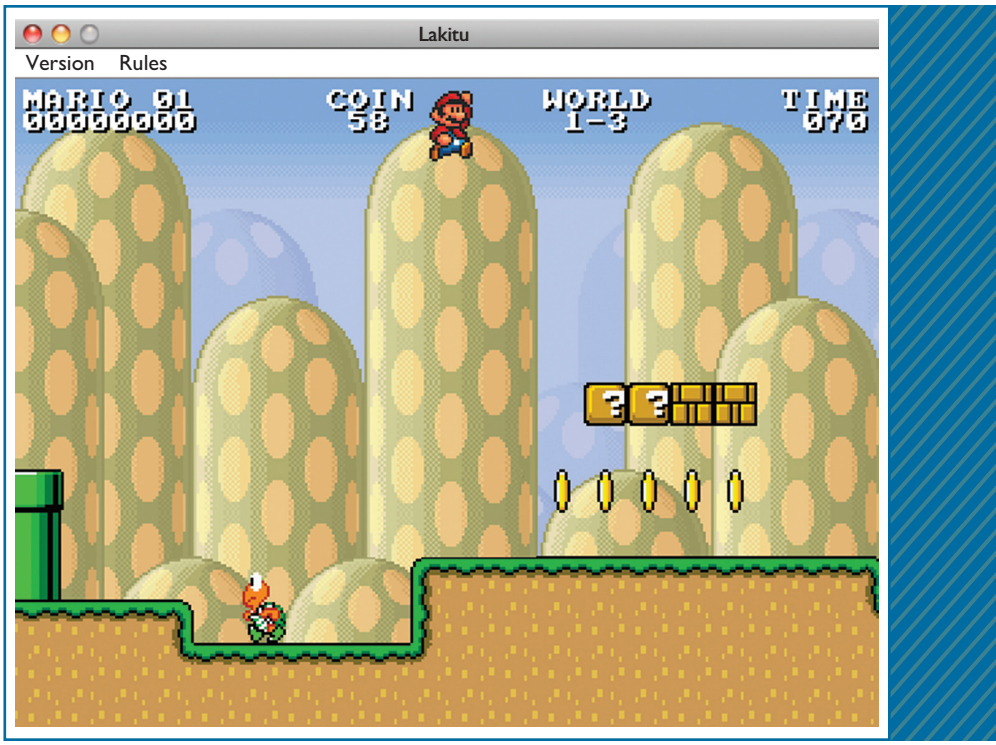

FIGURE 1. Lakitu in buggy operation, with the rule engine disabled. Notice how high Mario has jumped: a jump of such height should not be possible. Re-enabling the rule engine will prevent Mario from jumping so high again.

Attaching Mayet

Let's go through an example of a repair that Mayet can make if Mario jumps too high, as Figure 1 illustrates. It's easy to define a categorization of what should really happen: "Mario should not be able to jump more than five units high." The game must meet this specification. To send this information for the rule engine to monitor, we need to create an event when Mario jumps, with the jump height value attached. A value that is too high will break this specification. When the rule engine receives the jump event, it compares the in-game height with what it should be. Figure 2a shows how that rule could look.

Inside the `when` clause is a search for a jump event where the height is greater than five. If the rule engine finds such an event, it logs an error message and sends a repair message back. Not shown in the rule is a little bookkeeping that expires jump events so we don't continue evaluating the same jump over and over.

Let's consider a more difficult problem. Imagine Mario is hovering in the air. He jumps up to a normal height and just stays there, fluttering off the ground. Any single event doesn't give us enough information. A jump event only says how fast and how high Mario will go, but that's completely normal. Can this hovering be detected? The key is to realize that the state must be inspected over time, so we add a landing event, too. What goes up, must come down. Our rule needs to know about both the jump and the landing. Drools specifies temporal rules as shown in Figure 2b.

Drools evaluates this rule by first looking for a jump event. When it finds one, it waits to see if the next property

```
rule "marioJumpTooHigh"
  when
    Sjump : Jump(height > 5)
  then
    logger.info("Mario jumping too high");
    broker.send(newMessage(new MarioMovement(...)));
end
```

(a)

```
rule "marioHover"
  duration(2s)
  when
    Sjump : Jump()
    not(Landing(this after[0s,2s] Sjump))
  then
    logger.info("Mario was hovering");
    broker.send(newMessage(new MarioMovement(...)));
end
```

(b)

FIGURE 2. Examples of rules generated by the rule engine to create an event. (a) A rule specifying a maximum height for an event, and (b) a rule specifying a length of time between two events.

is matched—that a landing two seconds after the jump isn't found. If Mario hasn't landed, Drools sends a repair message asking that the game accelerate him down, just as before. Expressing rules temporally is very powerful, and lets us specify many useful and interesting properties about a game, be it a short-lived one like this or a property spanning the entire game.

We believe this syntax is far easier to read than many programming languages. Thus, the rule file can be handed to the game designer for validation, ensuring that faults aren't accidentally introduced, edge cases aren't missed, and the designer's intentions are met correctly.

Enacting Repairs

The rule engine communicates via messages. It never actually modifies the SUT's state. What happens to those messages? Lakitu listens for messages as part of its run loop. If a new repair message is waiting, it reads the type of message and its parameters, then enacts the fix. This is, by far, the most invasive change we made to Lakitu. To consume repairs, we modified the loop so that after the logical state is modified, the game listens for repair messages and enacts them. Because Lakitu runs asynchronously, these repair messages aren't guaranteed to correspond with the logical state change in that loop's execution.

We implemented functionality for repairs that could move Mario, add new level components, or remove enemies. One useful aspect of having the SUT enact the repair, aside from the independence issues mentioned previously, is that the SUT can choose when a repair should occur. Some repairs could be delayed until they're outside the player's cone of view or ignored if they're no longer relevant.

With repairs in place, we created a system that can recover from failures: it can monitor for problems, react to them, and dynamically repair them, all within a few frames. By comparison, Lakitu lets users turn bugs on or off and turn the rule engine on or off.

Limitations

The Mayet architecture generally focuses on repairing logical state. However, if there's a problem with how that state is represented—say, a bug in the graphics pipeline that results in screen corruption—Mayet can't repair (or perhaps even detect) the problem. This corresponds to the invalid graphical representation category in our failure taxonomy. Further investigation in this area might well yield a modification to the Mayet architecture that would be more suitable for this class of issue.

A common concern raised by those who see the Mayet architecture is the possibility that the system could get stuck in a repair cascade: one repair results in the violation of a different rule, which in turn issues a repair that violates a different rule, and so on. The worst-case scenario would be a repair loop, where repairs are constantly issued. Related to this is the concern that repairs result in something even more negative, such as a hovering character being accelerated into a spike pit.

Certain design strategies, such as making the minimum viable change in a repair and keeping rules as constrained as possible to avoid broad matches, will help. However, the only way to truly assert whether this is a problem, and how it can be solved, is to try and scale up to a full game, which we haven't yet attempted.

Although emergent software is generally unverifiable, it doesn't mean that we can't take steps to ensure positive user experiences. Games, like many user-oriented software products, have an acceptability bound. Using a system such as Mayet provides developers the means to ensure that their games stay within that bound, but bugs can still occur.

As software across all domains begins to show greater complexity and thus greater emergence, the ideas and techniques we've explored here will become increasingly important. We believe games herald the beginning of developers letting go of the notion of being able to test everything upfront. We're moving toward runtime recovery being an essential component of modern software development. Although not being able to verify software can be worrying, having the tools to deal with that reality is rather comforting.

Lakitu is free and open sourced under a BSD-style license, and can be downloaded from its website at www.zenetproject.com/pages/lakitu.

References

1. H. Haken, "The Challenge of Complex Systems," *Information and Self-Organization*, Springer, 2006, chap. 1, pp. 1–35.
2. K. Salen and E. Zimmerman, *Rules of Play*, MIT Press, 2004, pp. 350–353.
3. K. Starr, "Testing Video Games Can't Possibly Be Harder Than an Afternoon With Xbox, Right?" *Seattle Weekly*, 11 July 2007; www.seattleweekly.com/2007-07-11/news/testing-video-games-can-t-possibly-be-harder-than-an-afternoon-with-xbox-right.php.
4. J.C. Mogul, "Emergent (Mis)Behavior vs. Complex Software Systems," *SIGOPS Operating Systems Rev.*, vol. 40, no. 4, Apr. 2006, pp. 293–304.
5. A. Avižienis et al., "Basic Concepts and Taxonomy of Dependable and Secure Computing," *IEEE Trans. Dependable and Secure Computing*, vol. 1, no. 1, 2004, pp. 11–33.
6. J. Blow, "Game Development: Harder Than You Think," *ACM Queue*, vol. 1, no. 10, 2004, pp. 28–37.
7. D. Garlan, B. Schmerl, and S.-W. Cheng, "Software Architecture-Based Self-Adaptation," *Autonomic Computing and Networking*, Springer, 2009, chap. 2, pp. 31–55.
8. J.H. Perkins et al., "Automatically Patching Errors in Deployed Software," *Proc. ACM SIGOPS 22nd Symp. Operating Systems Principles* (SOSP 2009), ACM Press, 2009, pp. 87–102.
9. "Liveness Manifestos," Apr. 2004; www.cs.nyu.edu/acsys/beyond-safety/liveness.htm.

ABOUT THE AUTHORS

CHRIS LEWIS is pursuing his PhD as a member of the Software Introspection Lab, part of the computer science department at the University of California, Santa Cruz. His research interests include testing methodologies, data mining, and software modeling. Lewis has an MS in engineering from the University of Bristol. Contact him at cflewis@soe.ucsc.edu.

JIM WHITEHEAD is a professor in the computer science department at the University of California, Santa Cruz. His research interests include software evolution, software bug prediction, procedural content generation, and level design of computer games. Whitehead has a PhD in information and computer science from the University of California, Irvine. He is a member of the ACM and the International Game Developers Association (IGDA). Contact him at ejw@soe.ucsc.edu.