# Runtime Monitoring of Temporal Logic Properties in a Platform Game

Simon Varvaressos, Dominic Vaillancourt, Sébastien Gaboury, Alexandre Blondin Massé, and Sylvain Hallé*

Laboratoire d'informatique formelle
Département d'informatique et de mathématique
Université du Québec à Chicoutimi, Canada
shalle@acm.org

**Abstract.** We report on the use of runtime monitoring to automatically discover gameplay bugs in the execution of video games. In this context, the expected behaviour of game objects is expressed as a set of temporal logic formulæ on sequences of game events. Initial empirical results indicate that, in time, the use of a runtime monitor may greatly speed up the testing phase of a video game under development, by automating the detection of bugs when the game is being played.

## 1 Introduction

The domain of video games is currently booming; a recent Gartner survey revealed that consumer expenses for video games would raise from 67 billion dollars in 2011 to more than 112 billion by the year 2015 [2]. Similar to all computer systems, video games have not been spared from programming errors making their way to the release of a product. For example, in *Halo Reach* (2010), it is possible for players to go out of the game's map in some places, allowing them to make actions that would otherwise be forbidden [1].

It is therefore important for a designer to detect a maximum of gameplay errors as soon as possible during the development phase of a game, since for some systems, correcting an error using an update after the product's release is technically impossible. Moreover, video games are a special type of *emergent* system: their complexity arises from the combination of multiple simpler parts like the physics engine, the graphics or the graphical user interface. A minor problem can bring a bigger one later in the execution. Therefore, to facilitate debugging, it is important to identify exactly when a bug occurs and report it as fast as possible.

Typically, video game companies hire manual testers, whose hourly salary varies from $20 to $100, with the special purpose of discovering gameplay bugs and manually filing them into a bug tracker database. Obviously, this technique

* With financial support from the Natural Sciences and Engineering Research Council of Canada (NSERC) and the Fonds de recherche Québec – Nature et technologies (FRQNT).

A. Legay and S. Bensalem (Eds.): RV 2013, LNCS 8174, pp. 346–351, 2013.

is time-consuming and far from fail-proof: in some cases, gameplay bugs are not immediately apparent to the human eye. In this setting, the use of runtime verification techniques presents the potential for improving the gameplay bug harvesting step. However, video games rely a lot on fast player inputs and are much more sensitive to speed and timing than traditional software; it is therefore important that the use of a monitor does not slow down the game in any noticeable way. This paper presents early results on this approach and illustrates how gameplay bugs of a popular platform game, *Infinite Super Mario Bros.*, can be specified as temporal logic formulæ and efficiently caught at runtime using an off-the-shelf monitor.

## 2 Gameplay as Temporal Logic Constraints

As opposed to most standard software, video games are not entirely driven by the user. Most games include a physics engine and a form of artificial intelligence to update the game environment even in the absence of any input from the player. Moreover, these updates must be executed a minimum of 30 times (called *frames*) per second, with 60 frames per second (fps) being a reasonable target for quality animation. Noticeable disruptions of the frame rate are regarded by players as bugs and have in the past caused the demise of some video game titles. This concept is best exemplified by a well-known game called *Infinite Mario Bros.* (Figure 1), an open-source reimplementation of the popular platform game *Super Mario World*, where various enemies and other game objects move around the game area independently of the player's (i.e. Mario's) actions.

Infinite Mario is made of 6,500 lines of Java code and is available online.[1] It is notable for being the subject of many research works on game testing and applications of Artificial Intelligence algorithms in the past [5]. It has recently been used as a testbed for the automated application of condition-action rules aimed at correcting erroneous game states [7]. A similar approach has been applied to *FreeCol*, a free version of the strategy game *Civilization* [4].

In the following, we push the concept further and attempt to formalize the expected behaviour of various game objects as temporal logic constraints. In this context, events represent various changes of state, both of the player's character and of the surrounding enemies and objects. Each event is represented as a list of parameter-value pairs, and has a parameter called `name`, indicating the type of the event (e.g. Jump, Stomp, EnemyDead, etc.). The number and name of the remaining parameters may differ depending on the event's type. For example, when Mario stomps on an enemy, the unique ID of that enemy will be included in the event; when Mario jumps, the height of the jump will be recorded.

The rules used to express the properties to monitor are represented with LTL-FO$^+$, an extension of Linear Temporal Logic (LTL). For example, the following expression indicates that globally, if an enemy gets hit by a fireball that Mario

[1] `http://mojang.com/notch/mario/`

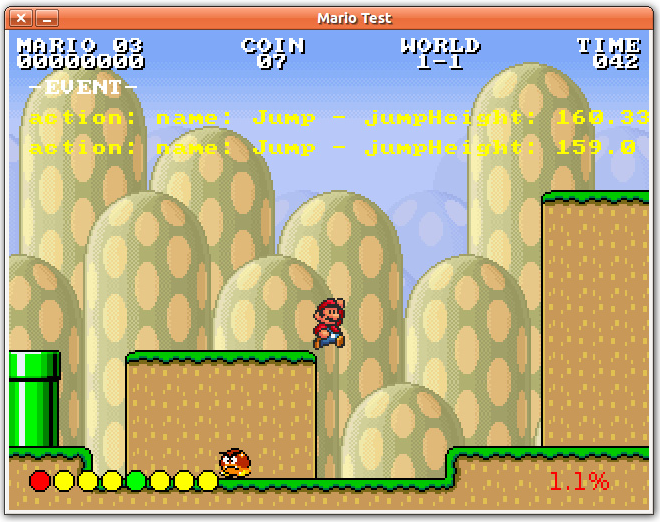



**Fig. 1.** The GUI of the modified version of Infinite Mario Bros

threw (event *name* EnemyFireballDeath), the next event should indicate the disappearance of the fireball so that it does not hit anything else.

$$\mathbf{G}\,(\text{name} = \text{EnemyFireballDeath} \rightarrow \mathbf{X}\,\text{name} = \text{FireballDisappear})$$

The presence of first-order quantifiers is necessary for two reasons: the same parameter may occur multiple times in the same event (such as when multiple enemy IDs are killed by Mario at the same time), and some gameplay properties may affect a single element across multiple events, as in the following expression.

$$\mathbf{G}\,((\text{name} = \text{Stomp} \wedge \text{isWinged} = \text{true}) \rightarrow \forall x \in \text{id} : \mathbf{X}\,(\text{id} = x \rightarrow \text{name} \neq \text{EnemyDead}))$$

In a normal playthrough, if Mario jumps on a flying enemy, it should lose its wings. In this formula, we make sure that a winged enemy cannot die after Mario stomps on it, as it should only stop flying. Since we are keeping the corresponding `id` in a variable named $x$, we can check the next event related to the *same* enemy to make sure it's respecting the normal flow of the game. However, one can see that, for this correlation between object IDs to be possible, first-order quantification on event parameters is necessary. As a matter of fact, we discovered early on that *propositional* Linear Temporal Logic is not expressive enough to represent but the simplest gameplay properties.

## 3 Empirical Results

Once a number of game properties were formalized as LTL-FO$^+$ formulæ, we devised an experimental setup to assess the performance of our runtime monitoring approach in actual runs of the game.[2] As we have seen, any errors caught by a monitor should be identified before the next frame, yielding an upper bound of 17 to 33 ms for the processing of each batch of events. Any processing time slower than this would either slow down the game and cause jerky animation, or have the monitor increasingly lag on the current game state and fill some event buffer.

The BeepBeep runtime monitor[3] [3] was selected to be inserted into the game, since it was developed in Java and uses LTL-FO$^+$ formulæ as its input language. The BeepBeep monitor accepts events in the form of XML strings. Some strings are constant, while others like this one are dynamically created based on the specific parameters of the event (enemy IDs, etc.). For example, the following shows the instrumentation to generate an event indicating that some enemy died:

```
MonitorTimer.Instance().updateWatchers("
 <action>
  <name>
   EnemyDead
  </name>
  <id>
   "+id+"
  </id>
 </action>
");
```

The game's code was manually instrumented to produce these events; about 30 locations in the code had instruction of this kind inserted. We could have chosen AspectJ [6] to facilitate the instrumentation but we decided not to because this solution is Java-specific, and most games use languages like C++ or even unique ones like UnrealScript. Relying on AspectJ would not faithfully represent the restrictions one shall face when monitoring video games in general.

To keep track of the different outcomes for each property, we also added some elements to the game's GUI. First, circles of colour, each representing a property, can be found on the lower left part of the screen. A green dot indicates a property evaluates to true on the sequence of events received so far, while red indicates it evaluates to false. Since each monitor is constantly queried on a finite trace prefix, the value of some properties may not be defined yet; this is indicated by a yellow dot. For debugging purposes, we also print the last two events produced at the top of the window. The lower-right corner displays in real time the overhead incurred by the presence of the monitors.

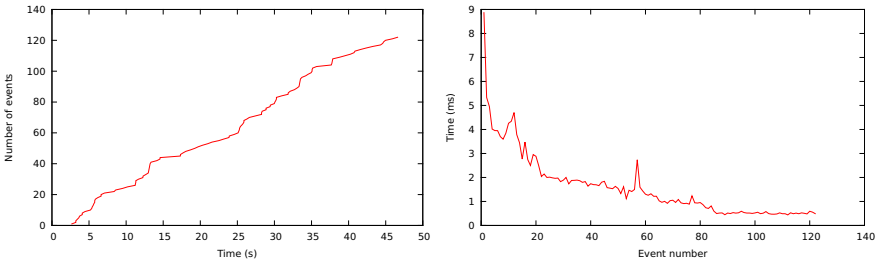
Finally, in order to make sure that our monitor can actually intercept gameplay bugs, we manually performed modifications to the game's code to create

[2] The instrumented version of Inifinite Mario and the runtime monitor can be downloaded from `http://github.com/sylvainhalle/BeepKitu`

[3] `http://beepbeep.sourceforge.net`

specific problems, such as removing instructions that handle the killing of some enemies. We then performed numerous runs of the game and computed various metrics on the game's and the monitor's execution.

The results were positive. Every formula we used could be monitored using our method, without slowing down the game in any noticeable way. A surprising finding of our study is that in a normal playthrough, the game generated roughly 2.9 events per second (Figure 2a). This event rate is very small compared to rates in typical runtime verification works. One can see that an event could take 9 milliseconds to process for 10 different properties (Figure 2b). It is possible to see a drop in the time required as the game progress, since monitors for properties that evaluate to true or false no longer need any updating. Even if, in a worst case scenario, no properties were resolved and each event took 9 milliseconds to handle, we can safely assume that it would not affect gameplay since 27 spare milliseconds are left before reaching the threshold time for 30 fps.




(a) (b)

**Fig. 2.** Experimental results on the monitoring of Infinite Mario Bros. a) Number of events generated in a sample run of the game; b) Monitor processing time for each event.

Finally, since video game companies keep track of their bugs in a database, we implemented a similar functionality using a MySQL server. We instrumented the code in such a way that, upon violation of some temporal property, the monitor sends an SQL query to the server, thereby automatically filing various metadata about the discovered violation: name of property violated, occurrence time, event trace prefix leading to the violation.

## 4 Conclusion

Overall, the results we obtained were conclusive as a first step in the application of runtime monitoring to video games. We succeeded in the monitoring of different properties using LTL-FO$^+$ in a video game without affecting the game experience. We also provided the game with a GUI that easily shows the outcomes for each monitored property. If one of them becomes violated, indicating a problem in the expected gameplay, the monitor automatically saves information about the bug in a database, something that could help in video game development.

Some improvements to the method could be implemented. For example, manual instrumentation of the game is tedious and error prone; it is believed that one could make good use of the *game loop* present in every video game to simplify its instrumentation: instead of manually finding and inserting the events to monitor, one could keep track of the game objects' state by interpreting the differences from one game loop iteration to the next. Moreover, compiling the monitor within the game does not seem a desirable choice for a larger-scale application of monitoring, as one would have to change the monitor to fit every game's implementation language. Finally, the game's graphical API made it hard to integrate monitor controls within its own GUI and limited the amount of information that could be input from (or displayed to) the user. As future work, we are currently working on a much larger open source game, drawing from the lessons learnt when monitoring Infinite Mario Bros.

## References

1. Worst videogame bugs of all time: From game-ending glitches to data-destroying nightmares, `http://bit.ly/GLySq`
2. Biscotti, F., Blau, B., Lovelock, J.-D., Nguyen, T.H., Erensen, J., Verma, S., Liu, V.K.: Market trends: Gaming ecosystem. Technical report, Gartner Group. Report G00212724 (2011)
3. Hallé, S., Villemaire, R.: Runtime enforcement of web service message contracts with data. IEEE Trans. Services Computing 5(2), 192–206 (2012)
4. Hamann, L., Gogolla, M., Kuhlmann, M.: OCL-based runtime monitoring of JVM hosted applications. ECEASST 44 (2011)
5. Karakovskiy, S., Togelius, J.: The Mario AI benchmark and competitions. IEEE Trans. Comput. Intellig. and AI in Games 4(1), 55–67 (2012)
6. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: Getting started with AspectJ. Commun. ACM 44(10), 59–65 (2001)
7. Lewis, C., Whitehead, J.: Repairing games at runtime or, how we learned to stop worrying and love emergence. IEEE Software 28(5), 53–59 (2011)