2014 IEEE International Conference on Software Testing, Verification, and Validation

Automated Bug Finding in Video Games: A Case Study for Runtime Monitoring

Simon Varvaressos, Kim Lavoie, Alexandre Blondin Massé, Sébastien Gaboury, Sylvain Hallé
Laboratoire d'informatique formelle
Département d'informatique et de mathématique
Université du Québec à Chicoutimi, Canada
Email: shalle@acm.org

***Abstract*—Runtime verification is the process of observing a sequence of events generated by a running system and comparing it to some formal specification for potential violations. We show how the use of a runtime monitor can greatly speed up the testing phase of a video game under development, by automating the detection of bugs when the game is being played. We take advantage of the fact that a video game, contrarily to generic software, follows a special structure that contains a "game loop"; this game loop can be used to centralize the instrumentation and generate events based on the game's internal state. We report on experiments made on a sample of five real-world video games of various genres and sizes, by successfully instrumenting and efficiently monitoring various temporal properties over their execution —including actual bugs reported in the games' bug tracking database in the course of their development.**

I. Introduction

The domain of video games is currently booming and is the sector of industry experiencing the fastest growth in the world [1]. 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].

While video games are regularly dismissed as mere entertainment, their complexity is on par with other modern computer systems and software. Consequently, they 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 [3]. In late 2011, Nintendo admitted that a bug in the latest release of its *Zelda* series prevented players from finishing the game when a precise sequence of actions is triggered [4]. It is therefore important for a designer to detect a maximum of programming errors as soon as possible during the development phase of a game, since for some systems, it is impossible to correct an error using a patch after the product's release. In any case, software bugs are frustrating for players, costly to game publishers and harmful to their reputation. One can therefore understand the interest in research and development of testing and analysis techniques adapted to the specific context of video games.

In this paper, we show how runtime monitoring can be used as an alternative to classical testing approaches used by most video game studios. In this setting, described in Section II, *properties* about the expected behaviour of the game are expressed formally in some language, and a runtime monitor observes the execution of the game and automatically notifies the user of the violation of these properties. Case in point, in Section III we describe five open source games of various sizes and genres, for which we enumerated gameplay properties.

In Section IV, we describe a runtime monitoring architecture for video games. This architecture is a departure from existing work on monitoring in many aspects. First, while known techniques for program monitoring insert numerous instrumentation points scattered throughout the code and corresponding to the occurrence of specific actions (function calls, etc.), we leverage the fact that most video games run what is called a *game loop*. Traditional instrumentation is replaced by a single insertion into the game loop, which generates an event filled with data from the game's current state. Monitoring properties on the game's execution must hence be expressed as temporal constraints over a succession of such "game snapshots". We shall see that this shift from an "event-based" semantics to a "snapshot-based" semantics has some implications over the language used to express the properties to monitor. Our architecture also provides a clean separation between the game and the monitor by having both reside in independent processes that communicate through the exchange of XML messages.

Section V experimentally evaluates the approach; it shows that complex properties on a large number of game objects can be enforced in real time at event rates exceeding 100 Hz. Most importantly, our experiments also demonstrate that we can successfully formalize and automatically detect gameplay bugs taken from actual bug reports filed by users into the games' bug trackers.

To the best of our knowledge, the monitoring approach we put forward in this paper is novel. Although manual testers are still required to operate the game, the detection and filing of bugs can be done automatically. It presents the significant appeal of extending existing works on runtime verification to greatly speed up the process of finding bugs during video game development.

II. Finding Bugs in Video Games

As we have seen, errors in a video game can lead to unacceptable behaviour of the software during its execution, and even cause unexpected interruption of the program. This section presents various ways to address the problem.

978-0-7695-5185-2/14 $31.00 © 2014 IEEE
DOI 10.1109/ICST.2014.27

A. Manual Testing

A widely accepted way to ensure the quality of a video game is to perform a series of rigorous tests throughout the development process. Some of the tests regard the usability of the game [5], while some others attempt to detect gameplay problems. With traditional testing methods, game programmers usually don't test their own games, nor have the time to do it. They only test each piece of their code before integration with the rest of the game.

It is only at specific phases in the game development process that versions of the game are passed on to external game testers [6], whose hourly wage can vary between $20 and $100. Each tester playing the game must manually file in a report for any erroneous behaviour observed, describing the conditions in which the error occurred and the consequences of that error. Those bugs are then stored into a bug database and addressed by programmers based on their severity.

This manual approach presents several drawbacks. First, it is time- (and money-) consuming, requiring the manual intervention of human testers at various points during the development process; this limits the number of times each test can be repeated. Second, a tester can only verify what is directly observable through the game's graphical interface, and what to be looked for may be very subtle (position of a character, wrong sprite used for some element, etc.). Finally, an error may only be detected if the tester recognizes it as such: some misunderstanding as to the game's expected behaviour may lead to erroneous behaviours go unnoticed.

B. Runtime Monitoring

Runtime monitoring is the process of observing an actual run of a system and dynamically checking and enforcing constraints on its execution [7]. This mechanism is more powerful than the mere verification of pre- or postconditions inside function bodies, as dependencies between *sequences* of method or function calls can be expressed and enforced. In the network protocols community, the technique has also been called *passive testing* [8]–[10].

Recently, automated monitoring of constraints has been implemented, on a small scale, to *Infinite Mario Bros.*, an open-source reimplementation of the popular platform game *Super Mario World* [11]. Properties on the gameplay were expressed as rules that were constantly evaluated during the execution. For example, one such rule expressed the fact that the player's character (Mario) cannot jump for a prolonged period of time (more than two seconds), and another one stated that Mario's jump height could not exceed five units. Should one of these rules be violated, the rule engine would alert the user and, in some cases, attempt to modify the game's state to revert it to a consistent set of values. A similar approach has been applied to *FreeCol*, a free version of the turn-by-turn strategy game *Civilization* [12].

All these solutions, while showing the potential for runtime verification in video games, are mostly *ad hoc* and very specific to the game being monitored. For runtime monitoring to have any chance of becoming an integral part of game testing, more general methods must be sought after.

In this respect, lessons can be drawn from past work on software runtime verification in general, where a large number of tools and approaches have been developed in the past decade. Notable monitor implementations include JavaMOP [13], LARVA [14], Tracematches [15], J-Lo [16], PQL [17], PTQL [18], SpoX [19], PoET [20] and TraceContract [21]. A large number of these works are implemented in Java and focus solely on the monitoring of method calls in Java programs. Most of them also require the instrumentation of the program required to relay the events to a monitor to be done through the use of aspect-oriented programming libraries, such as AspectJ [22] for Java. Finally, these monitors are generally compiled or weaved *into* the program, making them very coupled with the game to monitor. Consequently, they receive events as language-dependent pointcuts, and the specification of the properties to monitor refers directly to the program's objects.

Unfortunately, these assumptions no longer hold when the game to monitor is implemented in a different language as the monitor, which is bound to happen at some point. Indeed, game engines are available in a large variety of languages, including JavaScript (Gamvas[1]), Objective-C (cocos2d[2]), ActionScript (Flixel[3]), and Haskell (HGamer3D[4]); moreover, game publishers have started developing their own game languages, such as QuakeC [23] and UnrealScript [24].

C. Structure of a Typical Video Game

Even if most video games are different and each have their differences with others, they all follow some particularities. It goes without saying that video game programming is mostly object-oriented since most of the elements found in a game are objects. However, in opposition to regular software, a game is not completely guided by the user inputs. A software will not change state until the user wants to do something while a game will run even when the player is not doing anything. The game world is always changing or, at least, will always check if it needs to change anything, may it be the Non Player Characters' (NPC) artificial intelligence or physics. All of these actions are repeated every time the display is updated, i.e. upon each frame, in a centralized piece of code called called the *game loop*. In this loop, every logic that needs to be repeated, like physics of the game or the player inputs, will do so. Since every action is updated in the game loop, the state of every object of the game may be updated if necessary.

As a rule, a mechanism is present to regulate the frame rate so that the game runs at the same speed on every computer. There are two common ways to implement the frame rate: a *fixed* frame rate and a *variable* frame rate. In a fixed frame rate, the game loop is executed at a predetermined frequency. If the loop takes a shorter time, a sleep function is called to fill the time interval so that each turn of the loop has exactly

[1]http://gamvas.com/
[2]http://cocos2d.org/
[3]http://flixel.org/
[4]http://www.hgamer3d.org/

the same duration; the game will hence appear to "lag" when a computer cannot run the game loop fast enough.

On the contrary, in a variable frame rate, the game will adapt to the computer used by updating the game state by a time step commensurate with the elapsed time of the last iteration of the loop. Hence all events should take place at their usual speed, but their animation may appear jerky if not refreshed at a sufficiently high rate. In general, quality games attempt to refresh their display in the range of 30 to 60 frames per second (fps).

III. Motivating Examples

To illustrate the interest of monitoring techniques in the automated identification of gameplay bugs, our case study considers a sample of five open source video games.

1) *Pacman Canvas*[5] is a smartphone port of the popular arcade game Pacman; it is entirely made of HTML5 and JavaScript and runs in the FirefoxOS operating system.
2) *Infinite Mario Bros.*[6] is a platform game inspired from Nintendo's *Super Mario* franchise.
3) *Pingus*[7] is a clone of Psygnosis' *Lemmings* game series made by Ingo Ruhnke. It regularly counts among the highest quality open source games available and was once ranked in the Top 10 Linux games by CNN [25].
4) *Chocolate Doom*[8] is a port of the *Doom* first-person shooter published by ID Software. It is widely regarded as one of its most realistic clones, and even reproduces bugs and glitches of the original game.
5) *Bos Wars*[9] is an original, 3D real-time strategy game using the Stratagus[10] game engine. It is under active development since 2004 and has been part of the "42 More of the Best Free Linux Games" list by LinuxLinks [26].

Technical details about each game are given in Table I. Since our methodology requires access to a game's source code, only open source games could be considered for our study. Nevertheless, our selection covers five different types of games, four different programming languages, and a size in lines of code spanning two orders of magnitude. One can see that the palette of game genres selected covers 50% of all games sold in 2012. Moreover, it shall be noted that top-tier, or so-called "triple-A" video games (typically one order of magnitude larger) constitute a declining slice of the market, having experienced a 60% drop in the years 2009–2012 [27]). For these reasons, our selection of games makes a representative showcase for runtime monitoring.

Due to lack of space, we shall focus in this paper on Pingus to illustrate various elements of the proposed approach. However, the reader shall keep in mind that the same methodology has been successfully applied on all five games. Figure 1a shows a screenshot of version 0.7 of the game. The game is divided into more than 70 levels, each of which being populated with various kinds of obstacles, walls, and gaps. Between 10 and 100 autonomous, penguin-like characters (the Pingus) progressively enter the level from a trapdoor and start walking across the area. A Pingu keeps walking in the same direction until it either reaches a wall (in which case it turns around) or falls into a gap (and dies, if it falls from too high).

The goal of the game is to have a minimum percentage of the incoming Pingus safely reach the exit door. To this end, the player can give special abilities to certain Pingus, allowing them to modify the landscape in order to create a walkable path to the goal. For example, some Pingus can become Bashers and dig into the ground; others can become Builders and construct a staircase to reach over a gap. Other abilities modify the behaviour of other Pingus: hence the Blocker stands in the way and makes any Pingu that reaches it turn around as if it encountered a wall.

Each level starts with a fixed number of times each ability can be used; in some cases the puzzle is not so much to find a path to the goal, but rather to achieve it using the limited number of abilities made available to the player. Finally, to complexify things further, some surfaces (such as metal) cannot be broken through under any circumstances, and any Pingu reaching a body of liquid immediately drowns no matter what.

Since each Pingu is a simple, yet autonomous agent interacting with its surroundings, a large number of interesting properties regarding their behaviour can be expressed and monitored. We give some examples below:

P1. A Pingu on screen must be alive. Hence dead Pingus cannot clutter the screen and should be disposed of by the game loop.
P2. A Pingu cannot start exploding when it is falling. The only proper way for a Pingu to explode is a fixed amount of time after it has been assigned the task of Bomber.
P3. A Walker must always move. Pingus with no specific action must keep walking and cannot stall on the playing field.
P4. A Basher must stop bashing at some point in the game. Bashers dig horizontal tunnels into walls; all these walls are of finite dimensions and hence a Basher that keeps bashing until the end of a level indicates a problem.
P5. A Pingu falling faster than some predefined "deadly velocity" must keep falling, drown in water or splash on the ground. The fate of a Faller exceeding the velocity is hence sealed; for example such a Faller must not become a Walker again in the future.
P6. A Walker encountering a Blocker must turn around and keep on walking.

These rules are by no means a complete enumeration of the expected behaviour of Pingus. However, they provide a representative sample of the kind of gameplay constraints that can be violated due to bugs in the game under development. Ultimately, a large amount of the game's expected behaviour can be codified using rules of this kind.

[5] http://pacman.platzh1rsch.ch
[6] http://mojang.com/notch/mario/
[7] http://pingus.seul.org/welcome.html
[8] http://www.chocolate-doom.org/
[9] http://www.boswars.org
[10] http://www.stratagus.com

	Genre	Market share	Size (LOC)	Language
Pacman Canvas	Casual	3%	1.4K	HTML5/JavaScript
Infinite Mario Bros.	Platform	22%	6K	Java
Pingus	Puzzle	1.7%	40K	C++
Chocolate Doom	First-person shooter	21%	74K	C
Bos Wars	Real-time strategy	2.3%	113K	C++

TABLE I: A summary of the games surveyed in the case study. The market share figure is the one obtained from the 2012 sales of video games in the U.S. [27]; for example, 21% of all games sold were of the shooter genre.

(a) Pingus (b) Bos Wars

Fig. 1: Screenshots from two of the games included in our case study.

IV. A Monitoring Architecture

We present in this section an architecture for the automated monitoring of gameplay. This architecture will then be illustrated on Pingus; however, as we shall see, it is not tied to it and can easily be translated to other games implemented in various programming languages and game engines.

A. Desiderata and Design Decisions

Before setting up our monitoring architecture, we explain a number of design decisions that are motivated by technical constraints of the game environment.

1) Independent Monitor: First, the game and the monitor should be separate programs. As we have seen in Section II-B, this is a departure from most existing works in runtime monitoring. Apart from being tied to the implementation language of the game, the traditional process of compiling/weaving the monitor inside the system under test generally entails that the monitor processes the events synchronously: the generation of a new event passes the control to the monitor, and the execution of the program is resumed only when the monitor is done processing the event. In this context, placing the monitor in a separate thread within the program is a complex task, which on the other hand is completely taken care of by the operating system if the monitor is a program that runs on its own.

In addition, some user interaction is required with the monitor (to load the properties to monitor, display violations, etc.). Having the monitor run inside the game implies inserting appropriate controls inside the game's interface, using an API that will differ from one game to the next. Hence the whole monitor's GUI will need to be reprogrammed for every game, which would likely discourage practitioners from using it in the first place. Having the monitor lie in a separate program gives us complete control over its GUI, in which development time can be invested since it remains the same whatever the game under test.

2) Nested Event Structure: To properly enforce constraints P1–P6, one must be able to retrieve the action and position of every Pingu in the game, at every cycle of the game loop. The contents of each "event" generated must contain that information, and hence possess a nested data structure. However, by the remark above, we cannot pass native game objects directly to the monitor, as both are separate programs, and may even be implemented in different languages. We chose XML as an appropriate, language-agnostic format to serialize relevant values inside the game state.

3) Game Loop Instrumentation: Finally, as in any monitoring setup, the game must be *instrumented* in order to generate the events relevant to the properties to monitor and relay them to the monitor in some way. Aspect-based approaches are ill-suited for that task, as in many cases, the events associated to a property do not relate to a single function call, and hence cannot be easily defined through pointcuts. Moreover, an instrumentation solution tied to any particular programming language will leave out a large proportion of games and is hence deemed not desirable. Thus only manual instrumentation

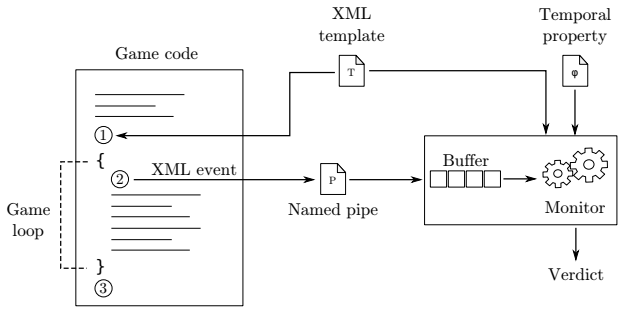

Fig. 2: A summary of the runtime monitoring architecture. Numbers 1–3 indicate the three locations where lines of code must be inserted to instrument the program. 1) Template instantiation; 2) Template rendering inside the game loop; 3) Template destruction.

is available. Yet, even for a simple game like Infinite Mario Bros., at least 30 different locations in the code would require the insertion of calls to the monitor. For Bos Wars, this number reaches an estimated 350 points.

However, we have seen in Section II-C that video games, fortunately, follow a particular structure where the bulk of the gameplay occurs through the repeated execution of a game loop. Each iteration of this loop modifies the state of the game in one atomic step; hence one can use the succession of game states as a surrogate for punctual events. For example, rather than look in the complete game's code for instructions that change the direction of walking of some Pingu (which can potentially be triggered from multiple locations in various files), one can instead detect such a change of direction by comparing two successive snapshots of the game's state and notice that the Pingu's x velocity changed its sign.

Our monitoring architecture will take advantage of this fact and generate events that are selective snapshots of the game's state at each cycle of the game loop. This presents the advantage that instrumentation, although manual, is kept to a minimum: once the game loop has been located, only this loop needs to be instrumented to produce events. This, in turn, entails that the properties to monitor, instead of being "event-based", will be expressed as constraints on the succession of game snapshots.

Figure 2 summarizes the runtime monitoring architecture. The monitoring process starts with a two-part specification: one the one hand, an XML *template* that will be used inside the game to periodically create an XML string, called an "event", based on data from the program's current state; on the other hand, a *temporal specification* expressing some desirable behaviour of the game in terms of a sequence of such events. A *named pipe* is used for passing events generated by the game to the monitor, which lies in a completely separate program.

Instructions are inserted into the game code in three locations, to retrieve the template, open/close the named pipe, and render the template and output the resulting XML string to the named pipe. On its side, the monitor continuously polls the named pipe, and processes the events against the temporal formula as they come. The named pipe is one-way: no feedback is injected back into the game when a violation is detected; this violation is rather displayed on the monitor's own GUI. As a matter of fact, the communication between the game and the monitor is completely asynchronous, so that the monitor's processing time for each event has no effect whatsoever on the game's performance (assuming they run in separate threads, which is generally the case in most multi-core computers).

In the following, we will present in detail a few elements of this architecture that require more explanations.

B. Serializing the Game's State with XML Templates

Rather than insert custom hand-written code fetching the game's current state to produce an XML event, we chose to further streamline the process and use an XML template engine. The principle of a template engine is simple: one first populates an associative data structure called a *data dictionary*. A template is then the definition of an XML document, into which template instructions are inserted. These instructions allow one to fetch values from the data dictionary, and to perform conditional evaluation and loops.

There exist many template engines; our choice fell on cpptemplate.[11] Table II shows an example of such a template for Pingus, written in cpptemplate. This template assumes that the data dictionary contains a field called `characters`, itself containing a list of characters called `pingus`. Line 3 of the template is enclosed in curly brackets, denoting a template instruction. It indicates that the contents of lines 4–11 will be repeated for every element inside this list; each element is put into a loop variable called `pingu`. It is then assumed that a `pingu` has fields `id`, `action`, `x` and `y`, whose values are inserted at specific locations in the document through template instructions on lines 5–8. Overall, rendering this template for a given data dictionary will produce an XML document listing the identifier and coordinates of all characters present in the game field at this particular moment.

The use of a form of template to serialize event data was advocated and tested in an embryonic manner in [28]. Its use over more hand- (or hard-) coded forms of instrumentation can be justified in many respects. First, it decouples event catching from property monitoring, making it an explicit two-stage process. The interception of events and their formatting into XML isolates the monitor from any language-dependent considerations.

Moreover, once the template rendering instructions are inserted in the game (as will be shown in the next section), one may use different templates to monitor different properties. In such a case, since the template is dynamically fetched from a file at runtime, the game itself does not need to be recompiled upon changing the template —thereby keeping a clear separation between templates and code.

Finally, since each processed method call created by the template is standard XML, the execution trace can easily be

[11] https://bitbucket.org/ginstrom/cpptemplate/wiki/Home

```
 1 <message>
 2  <characters>
 3   {% for pingu in characters.pingus %}
 4    <character>
 5     <id>{$pingu.id}</id>
 6     <action>{$pingu.action}</action>
 7     <isalive>{$pingu.isalive}</isalive>
 8     <position>
 9      <x>{$pingu.x}</x>
10      <y>{$pingu.y}</y>
11     </position>
12     <velocity>
13      <x>{$pingu.velx}</x>
14      <y>{$pingu.vely}</y>
15     </velocity>
16     <groundtype>{$pingu.groundtype}</groundtype>
17    </character>
18   {% endfor %}
19  </characters>
20  <overhead>{$overhead}</overhead>
21  <deadlyvelocity>{$deadlyvelocity}</deadlyvelocity>
22  <timestamp>{$timestamp}</timestamp>
23  <messagenumber>{$messagenumber}</messagenumber>
24 </message>
```

TABLE II: An XML template, written in cpptempl, that generates a message based on the game's current state.

recorded on disk and processed *post mortem*, even by a different tool, without losing any of the relevant information that was available at runtime.

C. Instrumenting the Game

The next step consists of adding code to the game so that state snapshots are produced on a regular basis. As we have seen in Section II, this is most efficiently done by locating the game loop, and adding instructions before, inside, and at the exit of that game loop. In the case of Pingus, this game loop is clearly found in the file `world.cpp`. However, the modifications to the characters are made in `pingu_holder.cpp` after the update method is called in the game loop; all the modifications we needed to make to the game were done in this single file.[12]

Table III illustrates the three points where modifications were made in the game. The first location is before the start of the game loop. In the case of Pingus, the most convenient location was into the constructor of `PinguHolder`. Lines 3–5 in Table IIIa were added, respectively to open the named pipe used to send events to the monitor (line 3), and to fetch the contents of the XML template from a temporary file (lines 4–5). Since an output stream was opened in the constructor, this stream must be closed upon exiting the game loop; Table IIIc shows the single instruction added in the destructor to this effect (line 3).

Now that the template has been properly fetched and that the output pipe is cleanly opened and closed, the bulk of the work can be added inside the game loop, in the body of method `update()`, as is shown in Table IIIb. An empty data dictionary is first instantiated (line 3). This dictionary is then filled in lines 4–7. The actual instructions that populate the dictionary with data have been omitted; they amount to looping through

[12]And in its companion header file `pingu_holder.hpp`, to add the declarations of variables `output` and `templ`.

```
1 PinguHolder::PinguHolder(const PingusLevel& plf) :
2 {
3    output.open("/tmp/monitor.pipe",
       std::ios::out);;
4    std::ifstream ifs("/tmp/monitor.template");
5    std::string templ(
       (std::istreambuf_iterator<char>(ifs)),
       (std::istreambuf_iterator<char>()));
6 }
```

(a) Constructor

```
 1 void PinguHolder::update()
 2 {
     ...
 3   cpptempl::data_map data_dict;
 4   PinguIter pingu2 = pingus.begin();
 5   while(pingu2 != pingus.end())
 6   {
       (Fill data_dict with attributes of each Pingu)
       ...
 7   }
 8   std::wstring result =
       cpptempl::parse(templ, data_dict);
 9   output << result;
     ...
10 }
```

(b) Game loop

```
1 PinguHolder::~PinguHolder()
2 {
3   output.close();
    ...
4 }
```

(c) Destructor

TABLE III: The instrumentation added to `pingu_holder.cpp` to generate events. The lines shown here are the only modifications made to the game.

each Pingu one by one, and pushing into the dictionary the values of its member fields.[13] Once the dictionary is filled, the template is rendered based on its contents (line 8), and the resulting XML string is output to the named pipe (line 9).

This also streamlines the modifications that must be made to the game. As a matter of fact, the instrumentation process described here was applied in a similar fashion for all games included in our case study, regardless of their size or their genre. For the fifth game of our case study (Bos Wars), the process had already become routine: identifying the game loop, inserting the template instructions and expressing properties was done in less than an hour. As long as the game loop can be identified, any game will be instrumented in the same way. Only the loaded template will differ to suit the particular objects present in the game's state, but this template is loaded at runtime anyway. Finally, since the monitor itself is an independent process, porting it to games developed in other languages simply amounts to implementing the XML template engine to that particular language —or using another engine, as long as it can produce XML strings by fetching the game's internal state.

[13]Note that method `update()` already loops through all Pingus on every call, so that the generation of the dictionary does not increase its complexity in this respect.

$$
\begin{array}{rcl}
\overline{m} \models x = y & \equiv & x \text{ equals } y \\
\overline{m} \models \neg\varphi & \equiv & \overline{m} \not\models \varphi \\
\overline{m} \models \varphi \wedge \psi & \equiv & \overline{m} \models \varphi \text{ and } \overline{m} \models \psi \\
\overline{m} \models \varphi \rightarrow \psi & \equiv & \overline{m} \not\models \varphi \text{ or } \overline{m} \models \psi \\
\overline{m} \models \mathbf{G}\,\varphi & \equiv & m_0 \models \varphi \text{ and } \overline{m}^1 \models \mathbf{G}\,\varphi \\
\overline{m} \models \mathbf{X}\,\varphi & \equiv & \overline{m}_1 \models \varphi \\
\overline{m} \models \varphi\,\mathbf{U}\,\psi & \equiv & m_0 \models \psi, \text{ or both } m_0 \models \varphi \text{ and } \overline{m}^1 \models \varphi\,\mathbf{U}\,\psi \\
\overline{m} \models \forall x \in \pi : \varphi & \equiv & \overline{m} \models \varphi[x/v] \text{ for all } v \in m_0(\pi)
\end{array}
$$

TABLE IV: Semantics of LTL-FO$^+$

D. Expressing Properties on Sequences of Game States

We now revisit the gameplay properties expressed earlier by writing them down as constraints on the sequence of game states. The language we present in this section is LTL-FO$^+$, a first-order extension of a well-known logic called Linear Temporal Logic (LTL), which has been shown to be appropriate for the modelling of so-called "data-aware" properties [29].

The building blocks for asserting properties over event traces are *atomic propositions*, which are of the form $x = y$, where x and y are either variables or constants. These atomic propositions can then be combined with Boolean operators $\wedge$ ("and"), $\vee$ ("or"), $\neg$ ("not") and $\rightarrow$ ("implies"), following their classical meaning. In addition, LTL *temporal operators* can be used. The temporal operator **G** means "globally"; hence, the formula $\mathbf{G}\,\varphi$ means that formula φ is true in every event of the trace, starting from the current event. The operator **F** means "eventually"; the formula $\mathbf{F}\,\varphi$ is true if φ holds for some future event of the trace. The operator **X** means "next"; it is true whenever φ holds in the next event of the trace. Finally, the **U** operator means "until"; the formula $\varphi\,\mathbf{U}\,\psi$ is true if φ holds for all events until some event satisfies ψ.

Finally, LTL-FO$^+$ adds *quantifiers* that refer to parameter values inside events. Formally, the expression $\exists x \in \pi : \varphi(x)$ states that in the current event m, there exists a value $v \in m(\pi)$ such that $\varphi(v)$ is true. The notation $m(\pi)$ denotes the set of element values in m matching some pattern π. Dually, the expression $\forall x \in \pi : \varphi(x)$ requires that $\varphi(v)$ holds for all $v \in m(\pi)$. When the context is clear, we abbreviate $\exists x \in \pi : x = k$ as π/k, stating that the (only) value at the end of path π is k. The semantics of LTL-FO$^+$ are summarized in Table IV.

The path expression π is written in a subset of XPath 1.0 [30]. A slash-separated list of elements denotes a sequence of nested elements in an XML structure; for example the expression `/a/b/c` denotes the value of all elements named `c` inside some `b` element, itself inside some `a` element at the root of the document. The double slash looks for all (and not only immediate) descendants of given name. Finally, the square brackets denote predicates, i.e. conditions that must apply to the element to be included in the selection. For example the path `//a[c=1]/b` fetches the value of all `b` elements nested in some `a` element, itself containing an element `c` with value 1.

Equipped with such a language, it is possible to revisit the monitoring properties expressed in Section III and write them as LTL-FO$^+$ formulæ. For example, Property P1 stipulates that all Pingus appearing onscreen must be alive. When a Pingu dies, its action status switches to value "dead". However, this Pingu must be discarded from the game state before the start of the next iteration of the game loop. Hence at any moment at the beginning of the game loop, the action state of every Pingu must not be "dead". This can be expressed by the following LTL-FO$^+$ formula:

$$\mathbf{G}\,(\forall x \in \texttt{//character/isalive} : x = \text{“true”})$$

This formula states that every event satisfies the following property: the value of element `<isalive>` inside every `<character>` element must contain the string "true".

Property P3 stipulates that a walker must move. Expressed in terms of game snapshots, this entails that the x position of the walker with given id c in two successive events must be different, which can be expressed in LTL-FO$^+$ as:

$$\begin{array}{c}
\mathbf{G}\,(\forall c \in \texttt{//character[action=walker]/id} : \\
\forall x \in \texttt{//character[id=}c\texttt{]/position/x} : \\
\mathbf{X}\,(\forall x' \in \texttt{//character[id=}c\texttt{]/position/x} : x \neq x')
\end{array}$$

It is here that the distinction between "event-based" and "snapshot-based" properties comes into play. If the game were instrumented in a classical way, all locations in the game's code where a Pingu is made to stop would be identified, and code producing an event saying "a Pingu has stopped" would be inserted in all these locations. In this setting, property P3 would be easy to monitor: it would suffice to raise a violation whenever such an event reaches the monitor.

However, in the present case, events are snapshots of the game's state, and the fact that some Pingu has stopped cannot be discovered by looking at a single event. Rather, one must correlate the x position of a Pingu in two successive events to find out about it. This, in turn, entails that such an x position of some event must be remembered, and compared with another value inside a subsequent event. Classical temporal logics, such as *propositional* LTL, cannot account for these comparisons. This is why LTL-FO$^+$, which supplies first-order quantifiers, is required in our monitoring setting. Therefore, the simplification of the game's instrumentation entails in counterpart a richer language to express the monitoring properties.

The remaining properties can be expressed as LTL-FO$^+$ expressions in a similar fashion. Their formalization is omitted due to lack of space.

E. Runtime Monitor

The particular monitoring engine used in this paper is BeepBeep, a Java-based monitor for web applications developed by the authors in previous work [29], and which has completely been rewritten as a stand-alone program for the needs of this experiment. The choice of BeepBeep can be explained by the fact that it is the only existing monitor able to monitor first-order temporal logic formulæ on XML events, although any other monitor with the same specifications could be used in its

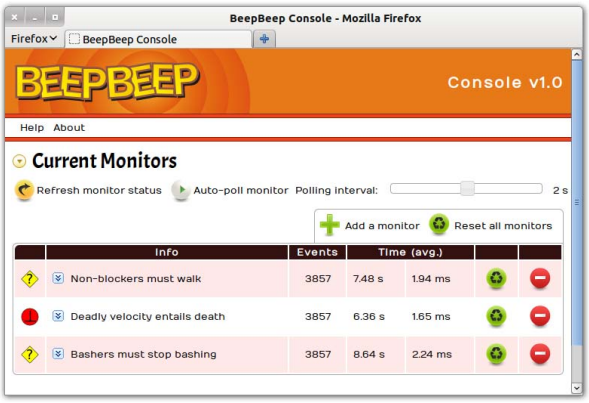

Fig. 3: The graphical user interface built around the BeepBeep monitor. Each property spawns one monitor instance, whose status is displayed in real time. Monitors can be added and removed on-the-fly while the game executes.

place. A graphical user interface has been built around it and is accessible from a web browser, as is shown in Figure 3.

For inter-process communication, we used the simple mechanism of a (Unix) named pipe. When first run, the monitor creates a named pipe in a fixed location (typically `/tmp/monitor.pipe`) using Linux's classical `mkfifo` command. Although it behaves like a regular file (and can hence be easily written to with a few lines of code in most programming languages), the operating system generally handles read/write operations on a named pipe into RAM, such that no slow disk access is ever required during the monitoring process. Moreover, as we have seen above, the events produced by the game are straightforward Unicode character strings forming an XML document, furthering their interoperability.

V. Experimental Results

To assess the feasibility of our approach, we performed an experimental evaluation of our monitoring setup on all five games listed in Table I. The complete runtime monitoring environment, including the instrumented source code of the games and code for the runtime monitor, is available online.[14]

For the first four games of our case study (Pacman Canvas, Infinite Mario Bros., Pingus and Chocolate Doom), only the most recent development snapshot was available, and the games' respective web site does not provide a link to a bug tracking database. Therefore, we purposefully inserted bugs into the games so that our monitor would actually need to catch them in a timely manner. These bugs were introduced by altering the game's source code; in the case of Pingus, they lead to the following undesirable behaviour:

- Some Pingus may explode when falling
- Some Pingus may remain on the game field after dying
- Some Pingus may enter the wrong state after a deadly fall
- Some Bashers may bash forever, even when there is no wall

[14]https://github.com/kimlavoie/BeepBeepPingus. The page provides links to instrumented versions of all games.

These bugs were selected because they can easily be seen on screen; it is hence easy to to see that when the bug occurs, the monitor reacts accordingly. Also, these bugs did not require complex changes to the source code that could have themselves impacted on the performance of the game; most of the time, we only had to change one or two lines in the game's code.

In the case of Bos Wars, both a history of development snapshots of the source code and a bug tracking database are available.[15] We proceeded to analyze the list of reported bugs over the course of the game's development and, out of the 36 entries it contains, compiled a list of bugs related to gameplay (as opposed to graphical, compiling- or animation-related bugs):

- 29095: Engineers autorepair a building that another engineer is recycling
- 29765: Jet fighters shooting each other
- 37861: "Set new units target" makes engineers harvest buildings

As was done for the other games, we succeeded in formalizing each of these bugs into an appropriate LTL-FO$^+$ formula expressed in terms of game objects. It was then possible to retrieve a snapshot of the game's code dating from the time the bugs were reported and use it as our "buggy" game.

We started each game and the monitor according to the setup shown in Figure 2, and proceeded to play them (manually) on various levels. Since each game introduces slight non-determinism, it was not possible to script the playing of a level so that it could automatically be replayed identically as many times as needed.

Every time a level was played, the complete trace of events sent to the monitor was saved to a file, as well as various statistics on the CPU time consumed by the game loop and the processing time on the monitor side. A manual inspection of the traces showed that they were very similar in terms of the parameters we collected. Therefore, in the following, we provide graphs for a single trace; the reader can assume that the trends they show apply in a very similar way to the remaining traces we collected.

The data was collected using a stock installation of Ubuntu 12.04.2 LTS on an Intel Dual Core P7350 2 GHz computer with 4 GB of RAM.

A. Event Count and Frame Rate

We first measured the cumulative number of events generated with respect to time for a representative run of the game. We observed that the number of events produced per unit of time is very constant throughout the course of every game; on our reference computer, it averages 167 events per second for Pingus. Pingus is programmed to run at a fixed frame rate of 40 fps, but updates the game state as often as it can. Hence a minimum of 40 events per seconds are required for the game not to lag. The above figure shows that the game had ample CPU time to update its state at more than four times that rate. This was the fastest refresh rate we observed; the remaining games had their game loop run in the range 40–60 Hz.

[15]http://savannah.nongnu.org/bugs/?group=stratagus-bos

On average, each event produced by our instrumentation results in an XML string of 700 bytes. We can hence estimate the bandwidth consumed by the game-to-monitor communication at about 120 kilobytes per second —a figure that could easily be cut in half by using shorthand names for XML elements and other trivial forms of compression.

B. In-Game Overhead

The second parameter we measured was the in-game overhead incurred by the presence of extraneous code inside the game itself. Apart from the instantiation and destruction of the XML template at the beginning and at the end of the game, all this overhead is caused by the lines of code 3–9 shown in Table IIIb. Since the playing had to be done by hand, we could not compare a trace without instrumentation with the same run of the game with instrumentation added. We therefore had to resort to indirect means of evaluating the overhead incurred by this instrumentation; we therefore measured the time elapsed executing these lines at each iteration of the game loop.

The elapsed CPU time increases in a roughly linear fashion, totalling 2 seconds elapsed time after 5,000 events, yielding an average of 0.4 ms per event. This shows that, on average, our reference computer could instantiate the XML template a maximum of 2,500 times per second. Since the observed event frequency (with instrumentation) is 167 Hz, we can deduce that the addition of a 0.4 ms delay to each event entails that the original event rate was probably around 179 Hz, yielding a 6.7% overhead. However, for a game running at exactly 40 events per second, adding 0.4 ms per event to render the template drops the frame rate to 39.4 fps, a reduction of only 1.5%.

C. Monitor Processing Overhead

We then switched to the monitor side and evaluated empirically its ability to process the stream of events produced by the game in a timely manner. As shown in Figure 2, the game-to-monitor communication is made through a named pipe, whose contents are is periodically polled by the monitor and stored into an input buffer. A monitor lagging on the game would be spotted through its input buffer filling regularly, and never returning to an empty state. Fortunately, we observed that the size of the input buffer, in our experiments, was always zero, indicating that the monitor could process every event as soon as it arrived.

This trend is confirmed by Figure 4, which shows the cumulative processing time on the monitor side for a representative run of the game. One can see that this cumulative time is linear, and hence that processing time per event is a constant value. For property P1, this processing time averages 0.13 ms per event, indicating that the monitor could process more than 7,500 events in a second.

However, property P1 is a simple one; we hence measured the processing time for a more complex property involving many quantifiers and temporal operators. Figure 5 shows the cumulative processing time of the monitor for property P5 (the most complex property included in our case study). This time, the processing time is less constant, with a spike of

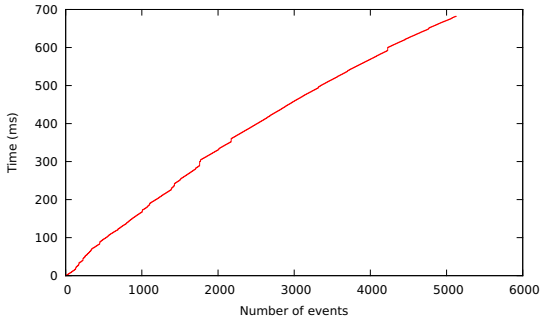

Fig. 4: Cumulative processing time on the monitor side for a run of the game on a simple formula (P1)

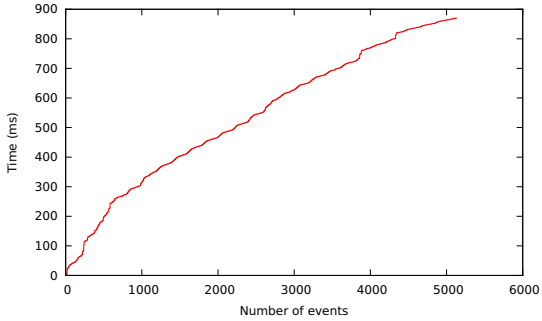

Fig. 5: Cumulative processing time on the monitor side for a run of the game on a complex formula (P5)

about 0.3 ms per event in the early phase of the game (first 1,000 events) and a shorter processing time in the remainder of the trace. The initial spike can be explained since property P5 expresses a fact about falling Pingus, and that in the beginning of a level, Pingus enter the map by falling through a trapdoor. However, when there are no Fallers, the quantifier $\forall c \in$ `//character[action=faller]/id` evaluates on the empty set, which results in a much faster processing of the formula.

D. Timely Detection of Bugs

The figures shown above indicate that the monitor could handle the real-time monitoring of the game's behaviour and had no trouble detecting the gameplay bugs when they occurred —including the user-reported bugs occurring in Bos Wars. In every game, watching the monitor's output provided immediate feedback on the occurrence of the bugs. On our reference computer, and assuming a conservative estimation that each property requires 0.3 ms per event to be monitored, the setup described above could monitor more than 50 properties of a complexity equivalent to P5 simultaneously at a frame rate of 60 fps. Since the generation of events inside the game is independent from their processing in the monitor, this limit only matters for the monitor not to fall behind with respect to the game's state, but has nothing do to with slowing down the game.

Moreover, we shall stress that the monitor software we currently implemented, although running in a thread separate from the game, is itself single-threaded. The implementation of multi-threading within the monitor could increase even further the number of properties that could be handled for a given frame rate by feeding the same event to multiple monitor instances in parallel.

VI. Conclusion

In this paper, we have presented a runtime monitoring architecture that allows the automated detection of gameplay bugs based on temporal logic specifications. Rather than relying on language-specific libraries, such as ApsectJ, to properly instrument the game's source code, we rather leveraged the fact that most video games contain a game loop where most of the processing takes place. The generation of events to be sent to an external monitor hence became centralized in a single location of the source code, which in turn simplifies the modifications required to the game for monitoring to take place.

Properties on the expected behaviour of the game then become constraints on possible successions of game state snapshots, whose content is automatically serialized with the use of a standard, XML templating library. The constraints are expressed in LTL-FO$^+$, an extension to Linear Temporal Logic allowing to correlate data parameters across multiple events of a trace. The monitoring setup was tested on five games of different genres, and demonstrated that complex properties (including real-world bugs reported by users) on a large number of game objects can be enforced in real time at event rates exceeding 100 Hz.

Based on the very promising results we obtained in our experiments, we are now working on refinements and additions to the proposed architecture. For example, work on the automated filing of property violations in a bug tracker is currently under way. It is also planned to replace clear-text XML used in the events with more compact forms of data used for networking.

Acknowledgements

The authors would like to thank the undergraduate students Raphaël Laguerre, Vincent Lavoie, Armand Lottmann and Dominic Vaillancourt who contributed to this research.

References

[1] TechnoCompétences, "L'emploi dans l'industrie du jeu électronique au Québec en 2010," Tech. Rep., October 2010.

[2] F. Biscotti, B. Blau, J.-D. Lovelock, T. H. Nguyen, J. Erensen, S. Verma, and V. K. Liu, "Market trends: Gaming ecosystem," Tech. Rep., 2011, report G00212724.

[3] , "Worst videogame bugs of all time: From game-ending glitches to data-destroying nightmares," http://bit.ly/GLySq.

[4] , "Game-breaking 'Song of the Hero' glitch," *IGN*, 2011, http://go.ign.com/xXqxoe.

[5] S. Laitinen, "Do usability expert evaluation and test provide novel and useful data for game development?" *Journal of Usability Studies*, vol. 1, no. 2, pp. 64–75, 2006.

[6] C. Schultz and R. Bryant, *Game Testing All in One*. Mercury Learning & Information, 2011.

[7] M. Leucker and C. Schallhart, "A brief account of runtime verification," *J. Log. Algebr. Program.*, vol. 78, no. 5, pp. 293–303, 2009.

[8] D. Lee, D. Chen, R. Hao, R. E. Miller, J. Wu, and X. Yin, "A formal approach for passive testing of protocol data portions," in *ICNP*. IEEE Computer Society, 2002, pp. 122–131.

[9] E. Bayse, A. R. Cavalli, M. Núñez, and F. Zaïdi, "A passive testing approach based on invariants: application to the WAP," *Computer Networks*, vol. 48, no. 2, pp. 235–245, 2005.

[10] H. Ural and Z. Xu, "An EFSM-based passive fault detection approach," in *TestCom/FATES*, ser. Lecture Notes in Computer Science, A. Petrenko, M. Veanes, J. Tretmans, and W. Grieskamp, Eds., vol. 4581. Springer, 2007, pp. 335–350.

[11] C. Lewis and J. Whitehead, "Repairing games at runtime or, how we learned to stop worrying and love emergence," *IEEE Software*, vol. 28, no. 5, pp. 53–59, 2011.

[12] L. Hamann, M. Gogolla, and M. Kuhlmann, "OCL-based runtime monitoring of JVM hosted applications," *ECEASST*, vol. 44, 2011.

[13] P. O. Meredith, D. Jin, D. Griffith, F. Chen, and G. Roşu, "An overview of the MOP runtime verification framework," *International Journal on Software Techniques for Technology Transfer*, 2011, to appear.

[14] C. Colombo, G. J. Pace, and G. Schneider, "LARVA — safer monitoring of real-time Java programs (tool paper)," in *Seventh IEEE International Conference on Software Engineering and Formal Methods (SEFM)*. IEEE Computer Society, November 2009, pp. 33–37.

[15] E. Bodden, L. J. Hendren, P. Lam, O. Lhoták, and N. A. Naeem, "Collaborative runtime verification with Tracematches," *J. Log. Comput.*, vol. 20, no. 3, pp. 707–723, 2010.

[16] V. Stolz and E. Bodden, "Temporal assertions using AspectJ," *Electr. Notes Theor. Comput. Sci.*, vol. 144, no. 4, pp. 109–124, 2006.

[17] M. C. Martin, V. B. Livshits, and M. S. Lam, "Finding application errors and security flaws using PQL: a program query language," in *OOPSLA*, 2005, pp. 365–383.

[18] S. Goldsmith, R. O'Callahan, and A. Aiken, "Relational queries over program traces," in *OOPSLA*, 2005, pp. 385–402.

[19] Ú. Erlingsson and M. Pistoia, Eds., *Proceedings of the 2008 Workshop on Programming Languages and Analysis for Security, PLAS 2008, Tucson, AZ, USA, June 8, 2008*. ACM, 2008.

[20] Ú. Erlingsson and F. B. Schneider, "IRM enforcement of Java stack inspection," in *IEEE Symposium on Security and Privacy*, 2000, pp. 246–255.

[21] H. Barringer and K. Havelund, "TraceContract: A Scala DSL for trace analysis," in *FM*, ser. Lecture Notes in Computer Science, M. Butler and W. Schulte, Eds., vol. 6664. Springer, 2011, pp. 57–72.

[22] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, "Getting started with AspectJ," *Commun. ACM*, vol. 44, no. 10, pp. 59–65, 2001.

[23] D. Hesprich, "QuakeC reference manual," Tech. Rep., 1998. [Online]. Available: http://pages.cs.wisc.edu/~jeremyp/quake/quakec/quakec.pdf

[24] "UnrealScript language reference." [Online]. Available: http://udn.epicgames.com/Three/UnrealScriptReference.html

[25] L. Anderson, "Top 10 Linux games for the holidays," 2000, http://archives.cnn.com/2000/TECH/computing/12/20/linux.games.idg/index.html. Retrieved March 28th, 2013.

[26] LinuxLinks, "42 more of the best free Linux games," Tech. Rep., 2009. [Online]. Available: http://www.linuxlinks.com/article/20080522164112313/Games-Part2.html

[27] NPD Group, "Essential facts about the computer and video game industry 2013," Entertainment Software Association, Tech. Rep., 2013. [Online]. Available: http://www.theesa.com/facts/pdfs/ESA_EF_2013.pdf

[28] J. Calvar, R. Tremblay-Lessard, and S. Hallé, "A runtime monitoring framework for event streams with non-primitive arguments," in *ICST*, G. Antoniol, A. Bertolino, and Y. Labiche, Eds. IEEE, 2012, pp. 499–508.

[29] S. Hallé and R. Villemaire, "Runtime enforcement of web service message contracts with data," *IEEE Trans. Services Computing*, vol. 5, no. 2, pp. 192–206, 2012.

[30] J. Clark and S. DeRose, "XML path language (XPath) version 1.0, W3C recommendation," 1999. [Online]. Available: http://www.w3.org/TR/xpath