UQAC

Université du Québec
à Chicoutimi

MÉMOIRE

PRÉSENTÉ À

L'UNIVERSITÉ DU QUÉBEC À CHICOUTIMI

COMME EXIGENCE PARTIELLE

DE LA MAÎTRISE EN INFORMATIQUE

PAR

SIMON VARVARESSOS

ÉTUDE DE FAISABILITÉ DU RUNTIME MONITORING DANS LES JEUX VIDÉO

JANVIER 2014

RÉSUMÉ

Dans le domaine des jeux vidéo, beaucoup de bogues se retrouvent sur le marché malgré les différentes phases de tests. Pour une industrie basée sur le divertissement, rencontrer des bogues qui empêchent le déroulement normal du jeu (dits « game breaking ») peut nuire à la vente du produit. Toutefois, la plupart des techniques d'assurance-qualité connues ne sont pas adaptées aux jeux vidéo. Il existe une technique de vérification, le runtime monitoring, qui observe l'exécution d'un système et pourrait améliorer la tâche d'élimination des bogues. Ce projet a pour objectif de prouver la faisabilité de l'utilisation d'une telle technique dans ce domaine.

Ce travail vise à améliorer le processus de recherche et d'élimination des bogues retrouvés dans l'entreprise ainsi qu'à agrandir le champ d'action des méthodes formelles. Pour ce faire, ce mémoire propose une nouvelle méthode d'instrumentation prenant en compte des caractéristiques propres aux jeux vidéo.

Ce mémoire ne représente qu'une première partie d'un projet plus grand visant non seulement à rendre possible l'utilisation du runtime monitoring mais aussi à améliorer le processus de détection des bogues en facilitant la tâche des testeurs et développeurs.

REMERCIEMENTS

Tout d'abord, je tiens à remercier mon directeur de maîtrise, M. Sylvain Hallé, qui a été présent tout au long de ce projet et ce, avec une disponibilité incroyable. Il n'a pas hésité à me donner des conseils pour le travail à effectuer, à m'aider dans l'écriture qui a parfois été difficile et en me donnant de nombreuses opportunités d'augmenter mon expérience dans la recherche. Il n'y avait aussi jamais de moment ennuyeux et l'ambiance était toujours des plus agréables. Il m'a aussi donné l'occasion d'enseigner durant mes études, ce qui fut une expérience plus qu'intéressante ayant contribué à mon désir de continuer mes études.

J'aimerais également remercier mon codirecteur de maîtrise, M. Sébastien Gaboury, qui était toujours présent lors des rencontres, contribuant énormément à l'avancement des idées et me donnant des conseils pour des conférences et des articles. Il a aussi toujours fait en sorte que les réunions soient amusantes. Je voudrais aussi mentionner M. Alexandre Blondin Massé qui a aussi participé aux réunions.

Je tiens à remercier le travail des différents étudiants de baccalauréat ou de maîtrise qui ont travaillé sur le projet, c'est-à-dire Kim Lavoie, Raphaël Laguerre, Vincent Lavoie, Armand Lottman et Dominic Vaillancourt.

De plus, je voudrais souligner le support de ma famille qui m'a permis de me concentrer entièrement sur ma maîtrise et qui a toujours cru en mes capacités.

TABLE DES MATIÈRES

TABLE DES FIGURES

LISTE DES TABLEAUX

CHAPITRE 1

INTRODUCTION

Depuis le début de leur existence, les systèmes informatiques ne cessent d'augmenter en taille et en complexité. Cette augmentation rapide a pour effet d'introduire de nombreux bogues qui survivent régulièrement jusqu'à la livraison du produit final. Un rapport sur le sujet indique qu'en 2002, les coûts annuels d'une mauvaise infrastructure de tests pouvaient s'élever jusqu'à 59,5 milliards de dollars (NIST, 2002). Avec des conséquences aussi élevées, il est donc primordial pour les entreprises de maximiser l'efficacité de leurs tests pour éviter cette perte d'argent. L'industrie des jeux vidéo ne fait pas exception à la règle. Le marché des jeux vidéo, évalué à 67 milliards de dollars en 2012 (Gaudiosi, 2012), occupe désormais une place très importante dans le domaine de l'informatique, mais les bogues nuisent, là aussi, beaucoup aux entreprises. Dans *The Legend of Zelda : The Skyward Sword*, un jeu faisant partie d'une des plus importantes franchises de Nintendo, une certaine combinaison d'événements faisait en sorte que le joueur ne pouvait plus continuer le jeu (Watts, 2012). Dans le même article, il est question d'autres problèmes, allant des plus inoffensifs, comme celui de *Fallout : New Vegas* où il était possible de trouver les intestins des ennemis flottant dans les airs, à d'autres plus sérieux, comme la corruption des données sur la carte mémoire dans le jeu *Soul Calibur 3* qui amenait la perte des sauvegardes de tous les jeux de l'utilisateur. Dans un domaine où les ventes sont basées sur le divertissement, les différents bogues, surtout ceux qui sont dits

« game breaking », sont à éviter.

Étant très différents des logiciels traditionnels, les jeux vidéo sont difficiles à tester de la même façon. Le code source peut être simple ou complexe, mais ce qui amène la plus grande difficulté, c'est l'interaction entre les différents éléments du système et le joueur, ce qu'on appelle l'émergence. Mogul (2006) donne une définition de l'émergence pour tenter de différencier les bons et les mauvais comportements dû à ce phénomène : « ce qui ne peut pas être prédit par une analyse faite à un niveau plus simple que celui du système en entier ». Ainsi, puisque la plupart des méthodes de vérification automatique requièrent de connaître les différentes entrées et sorties possibles, à l'instar des tests unitaires (Board, 1987), et que l'émergence crée justement des entrées inconnues, ces méthodes ne peuvent pas permettre d'identifier toutes les erreurs pouvant se présenter (Lewis et Whitehead, 2011). C'est pour cette raison que les entreprises travaillant dans le domaine des jeux vidéo n'ont qu'une solution : engager des testeurs pour éliminer le plus grand nombre de bugs possibles. Il existe toutefois une méthode, le runtime monitoring, qui examine l'exécution d'un système afin de détecter les anomalies (Leucker et Schallhart, 2009).

Le runtime monitoring permet d'observer l'exécution complète d'un logiciel, ce qui permet de vérifier les problèmes causés par l'émergence. Cette technique apporte toutefois certains problèmes. Puisque nous observons un système durant son exécution, le runtime monitoring utilise des ressources qui sont partagées entre le moniteur, qui surveille le système, et, dans notre cas, le jeu vidéo. Il est toutefois important qu'il n'y ait aucun ralentissement lorsque le jeu est en cours puisque le joueur doit, dans la majorité des jeux, réagir rapidement aux différentes situations qui lui sont présentées. Il faut donc s'assurer que le moniteur ne consomme pas trop de ressources. L'implémentation de la méthode peut aussi sembler inadéquate comparativement à son usage habituel. Par exemple, une étape de l'implémentation, l'instrumentation, est souvent accomplie de façon manuelle qui demande trop de temps pour

un jeu d'une grande envergure ou en utilisant des outils comme AspectJ (Nusayr, 2008) qui ne sont pas compatibles avec tous les langages de programmation utilisés dans l'industrie. Il faut donc rechercher de nouvelles façons d'implémenter le runtime monitoring tout en s'assurant de respecter les contraintes amenées par les jeux vidéo.

L'objectif de ce mémoire est de démontrer la faisabilité du runtime monitoring pour la détection automatique des bogues rencontrés dans les jeux vidéo par une preuve de concept. Pour ce faire, il faut d'abord établir les critères imposés par les jeux vidéo. Il faut donc regarder les éléments importants reliés aux tests dans l'industrie et viser à satisfaire les exigences pour une bonne assurance-qualité. Par la suite, il faut représenter des propriétés du jeu sous une forme reconnue par le moniteur. Ce mémoire se concentre sur l'utilisation de la logique LTL (Linear Temporal Logic) couramment utilisée pour le runtime monitoring (Leucker et Schallhart, 2009). On applique par la suite cette méthode sur plusieurs jeux différents. Il est important de choisir des types de jeu différents afin de démontrer l'universalité de la technique sur une majorité des jeux existants. L'utilisation ne doit pas se limiter à un seul genre pour que le runtime monitoring soit considéré comme utile dans le domaine. Pour terminer, il faut étudier l'impact du fonctionnement du moniteur sur la performance du jeu surveillé. Ce dernier ne doit pas nuire à l'exécution du programme sous surveillance.

L'utilisation du runtime monitoring dans le domaine des jeux vidéo ne semble pas fréquente. Les travaux de Lewis et Whitehead (2011) démontrent la seule méthode étudiée à ce jour. Toutefois, plusieurs problèmes ressortent, surtout du côté de l'instrumentation. Faite de manière manuelle, elle peut être appropriée pour des jeux simples comme Infinite Mario Bros, surtout en ajoutant les bogues soi-même. Par contre, les jeux d'une plus grande envergure ne permettent pas d'examiner le code en entier. Par exemple, le jeu *Total War : Rome 2* contient 3 millions de lignes de code (Purchese, 2012), rendant la tâche pratiquement impossible. De plus, l'ajout de nouvelles spécifications amène l'ajout de nouveaux événements à produire,

forçant à retourner chercher dans le code. Il est donc intéressant de songer à une méthode moins intrusive diminuant non seulement le temps d'implémentation, mais permettant de limiter le plus possible les endroits à modifier dans le système à surveiller. Dans ce mémoire, une solution à ce problème est proposée.

Ce mémoire a été divisé en quatre autres chapitres séparant les connaissances requises importantes et le résultat de la mise en application de la solution.

Le deuxième chapitre décrira le fonctionnement des tests d'assurance-qualité dans le domaine des jeux vidéo. Les différents types de bogues rencontrés seront énumérés afin de pouvoir regarder les différentes propriétés qui peuvent être écrites. Les problèmes reliés à ces tests seront aussi mentionnés pour expliquer en détails en quoi le runtime monitoring peut améliorer les résultats.

Le troisième chapitre expliquera précisément ce qu'est le runtime monitoring et comment l'appliquer. Les différentes étapes d'implémentation seront décrites dans l'ordre ainsi que la façon de les appliquer dans les jeux vidéo. Le fonctionnement et la syntaxe du langage LTL, utilisé pour écrire les spécifications, seront expliqués. À partir de ces informations, il sera possible d'intégrer cette solution à notre champ d'intérêt.

Le quatrième chapitre comportera les détails de la partie expérimentale. Il y aura d'abord l'explication des différentes prises de décision. Les jeux et le moniteur choisis seront décrits et les raisons de leurs choix seront mises en évidence. Les différentes propriétés choisies seront transformées, par la suite, en formules logiques selon la syntaxe décrite au chapitre précédent. Les conditions des tests et les résultats de ceux-ci seront réunis au même endroit pour ensuite déduire l'efficacité de la solution.

La conclusion ramènera les points importants du mémoire et ce que cette solution apporte aux

domaines de la vérification de logiciels et des jeux vidéo. Puisque ce mémoire représente une preuve de concept, il y aura aussi l'énumération de certains éléments qui peaufineraient la méthode.

Le travail présenté dans ce mémoire a fait l'objet de deux publications scientifiques (Varvaressos et al., 2013) (Varvaressos et al., 2014).

CHAPITRE 2

LES TESTS DANS LES JEUX VIDÉO

Le domaine des jeux vidéo, bien que priorisant encore avant tout le divertissement, est devenu un sujet intéressant pour les chercheurs. Ce thème central permet de travailler sur différents aspects reliés à d'autres spécialités très variées. Certains travaux se concentrent sur l'utilisation de la réalité virtuelle (Steinicke et al., 2009), sur les possibilités scientifiques qu'offrent les périphériques d'entrée comme la *WiiMote* (Shirai et al., 2007), sur l'étude des comportements des joueurs (Moura et al., 2011) ou, comme le sujet de ce mémoire, sur les méthodes de vérification (Lewis et Whitehead, 2011). Cet intérêt de la communauté scientifique n'a pas toujours existé et les entreprises issues du domaine des jeux vidéo ont dû développer certaines techniques propres au domaine des jeux vidéo. Notamment, puisqu'elles travaillent sur des logiciels parmi les plus compliqués du domaine de l'informatique, elles ont développé une façon de travailler qui permet de détecter et éliminer une grande partie des bogues.

Ce chapitre commencera par une description des bogues qu'il est possible de rencontrer dans les jeux vidéo. Il existe plusieurs taxonomies des bogues pour les logiciels traditionnels, mais ces dernières sont souvent peu adaptées aux jeux vidéo. Nous présenterons la classification de Lewis et al. (2010) ainsi qu'une échelle de sévérité du problème qui est traditionnellement utilisée dans l'industrie. Par la suite, les méthodes et les outils utilisés pour l'assurance-qualité

du produit seront étudiés puisqu'il est important de prendre en compte le fonctionnement du milieu. La fin de ce chapitre amènera les points faibles des méthodes afin de justifier l'intérêt d'introduire une méthode alternative telle que le runtime monitoring.

2.1 LES BOGUES DANS LES JEUX VIDÉO

Bien que des bogues soient aussi présents dans les logiciels traditionnels, il est possible d'observer différents comportements lorsqu'ils appraissent dans les jeux vidéo. Les bogues ne sont pas nécessairement perçus de la même manière et il est donc important de bien définir le terme.

2.1.1 DÉFINITION D'UN BOGUE DANS LE DOMAINE DES JEUX VIDÉO

Levy et Novak (2010) décrivent un bogue dans les jeux vidéo de la façon suivante : « tout élément qui n'améliore pas le jeu et qui lui nuit est un bogue ». Ils précisent aussi qu'un bogue ne doit pas avoir été ajouté de façon planifiée (dans ce cas là, on parle d'un « Easter Egg » qui n'est pas important pour notre recherche). La taille de ce dernier n'a pas d'importance. Il est aussi important de séparer les problèmes qui sont liés à l'implémentation et à la spécification (Lewis et al., 2010). Les bogues d'implémentation proviennent d'un élément qui va à l'encontre de la spécification donnée dans le document de design du jeu alors que les bogues de spécification proviennent d'une erreur dans les choix des concepteurs, ce qui amène un comportement étrange. Il est possible que ce comportement soit en fait désiré ou, du moins, découle d'un élément désiré du jeu et il est donc difficile, la plupart du temps, de les considérer comme de vrais bogues à surveiller.

2.1.2 TERMINOLOGIE DES BOGUES

Lewis et al. (2010) formalisent le terme bogue en trois couches distinctives (faute, erreur et échec) en reprenant la terminologie d'Avizienis et al. (2004), mais en les adaptant spécifiquement aux jeux vidéo.

— **Faute :** Une faute est un problème qui a été ajouté lors du design ou de l'implémentation qui mène à une erreur dans le système. La faute est un élément dans le code qui déclenche le bogue.
— **Erreur :** L'erreur est la manifestation de la faute lors de l'exécution qui crée un état pouvant mener à un échec si les conditions se présentent.
— **Échec :** L'échec est une déviation du comportement attendu du système qui peut être observée par l'utilisateur. Ce dernier varie beaucoup et peut être très facile à voir ou, au contraire, presque invisible pour l'humain.

Prenons par exemple un bogue de *Dragon Age 2*, désormais réglé, qui rendait un personnage complètement inutile en combat (Bartel, 2011). À un certain point dans le jeu, le joueur pouvait gagner une habileté passive d'une alliée qui augmentait sa vitesse d'attaque lorsque cette dernière était dans la même équipe que le héros. Par contre, lorsqu'on la retirait de l'équipe, le jeu enlevait le bonus aux capacités initiales du personnage principal, baissant ainsi sa vitesse d'attaque au lieu de tout simplement désactiver l'habileté. Ce problème dans le code représente la faute. Lorsqu'on enlève le personnage, l'erreur se présente. L'échec obtenu est la baisse de puissance nette du personnage qui peut, à un certain moment, attaquer à une vitesse beaucoup trop lente qui ne respecte pas l'intention des concepteurs.

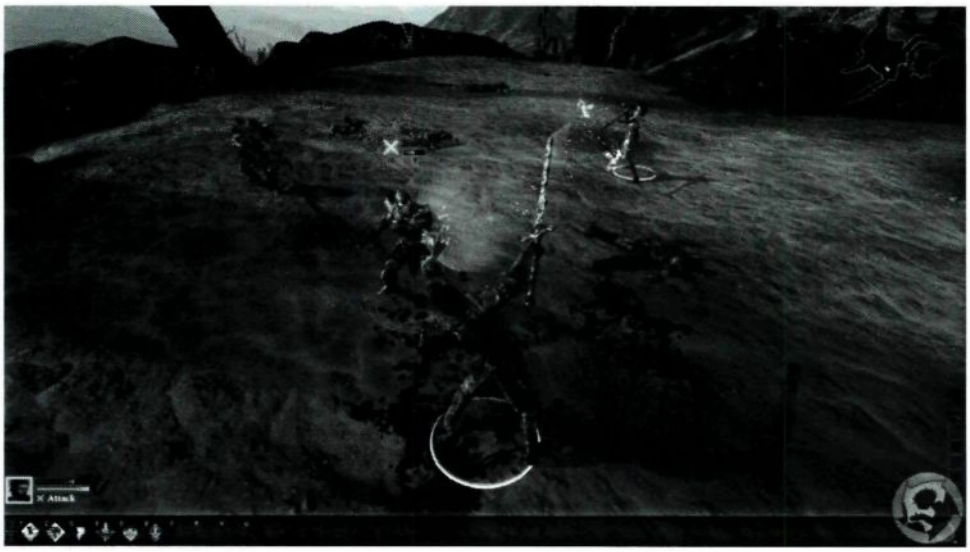


Figure 2.1: Capture d'écran du jeu *Dragon Age 2*

2.1.3 TAXONOMIE DES BOGUES

Avant de séparer les bogues en différents groupes, il est possible de remarquer deux grandes catégories d'échecs : les échecs temporels et les échecs non-temporels (Lewis et al., 2010). Les échecs temporels requièrent d'avoir certains renseignements sur l'état précédent du jeu pour être considérés comme des bogues alors que les échecs non-temporels peuvent être trouvés en regardant un seul état du système. Les auteurs ont remarqué que même si certains échecs se ressemblent, ils ne se détecteront pas nécessairement de la même manière. L'exemple utilisé prend en compte un personnage qui saute dans les airs. Si le personnage saute à une hauteur qu'il ne devrait normalement pas atteindre, il est possible de remarquer, en un seul point de l'exécution, qu'il a atteint une hauteur trop haute. C'est un échec non-temporel. D'un autre côté, si le personnage saute et reste coincé dans les airs à une hauteur normale, le problème n'est pas le même. Le personnage devrait normalement monter et redescendre, ce qui fait en sorte que malgré qu'il soit encore dans les limites de l'espace, il ne se trouve pas à une position valide. C'est un échec temporel. Le tableau 2.1 sépare chaque catérogie en plusieurs

groupes. Certaines catégories ont été fusionnées dans un article différent (Lewis et Whitehead, 2011).

Voici une brève description de chaque catégorie :

- **Position invalide :** Un objet est à un endroit où il ne devrait pas être.
- **Représentation graphique invalide :** Un objet n'apparait pas correctement ou n'apparait pas du tout.
- **Changement de valeur invalide :** Une valeur comme le pointage a été modifiée de façon incorrecte.
- **Stupidité artificielle :** L'intelligence artificielle a effectué une action qui ne doit pas être permise.
- **Mauvaise information :** Le joueur obtient une information qu'il ne devrait pas avoir, comme par exemple, la carte complète du niveau, n'a pas reçu une information qu'il devrait connaître ou reçoit de l'information qui n'est plus valide.
- **Mauvaise action :** Un personnage entreprend une action qui ne devrait pas être possible.
- **Position invalide à travers le temps :** Décrit un mouvement invalide qui ne provient pas nécessairement d'un déplacement vers un endroit qui n'est pas permis.
- **Contexte d'état invalide à travers le temps :** Un objet est dans un mauvais état pour une période de temps incorrecte.
- **Occurence d'événement invalide dans le temps :** Événement qui arrive trop souvent ou trop peu fréquemment.
- **Événement interrompu :** Un événement se termine avant d'atteindre la fin.
- **Temps d'implémentation :** Le jeu ne donne pas de réponse assez rapidement, communément appelé « lag ».

Tableau 2.1: Taxonomie des bogues selon Lewis, Whitehead et Wardrip-Fruin.

Échecs non-temporels	Échecs temporels
Position invalide	Position invalide à travers le temps
Représentation graphique invalide	Contexte d'état invalide à travers le temps
Changement de valeur invalide	Occurence d'événement invalide dans le temps
Stupidité artificielle	Événement interrompu
Mauvaise information	Temps d'implémentation
Mauvaise action	

Vijayaraghavan et Kaner (2003) ont démontré qu'il y avait de nombreux avantages à une telle séparation. Les informations obtenues d'une telle taxonomie permettent non seulement de séparer chaque bogue rencontré, mais permet aussi d'aider les testeurs à établir quels tests devraient être mis en place. L'utilisation d'une taxonomie augmente le nombre d'idées de tests intéressantes, permet d'obtenir des idées plus exhaustives et détaillées et diminue la perte de temps lors du brainstorming.

2.1.4 SÉVÉRITÉ DES BOGUES

Plusieurs bogues se retrouvent dans le produit final. En fait, les entreprises du domaine des jeux vidéo sont souvent très sévères au niveau des dates limites pour sortir un jeu et seuls les cas critiques peuvent retarder sa sortie. On sépare alors les bogues en quatre catégories différentes selon la sévérité de chacun d'entre eux pour s'assurer que les plus importants ne se retrouvent pas sur le marché (Levy et Novak, 2010).

— **Priorité faible :** Les bogues de cette catégorie ne font aucune différence pour l'équipe de développement, qu'ils soient réglés ou non. Ces derniers peuvent varier de petites erreurs graphiques ou sonores à des erreurs grammaticales dans l'interface. Les échecs qui n'ont aucun impact sur le déroulement du jeu ou sur la perception du joueur sont

de cette catégorie. Ils ont de bonnes chances de se retrouver dans le produit final puisqu'ils n'ont pas de réel impact négatif.

— **Priorité moyenne :** La différence entre cette catégorie et la précédente repose davantage sur la fréquence d'apparition. Encore une fois, il est souvent question de problèmes graphiques ou sonores, mais qui arrivent régulièrement. Ce sont des échecs qui ne nuisent pas au déroulement du jeu, mais qui peuvent être énervants aux yeux de l'utilisateur. Ces bogues sont souvent réglés vers la fin de la période de tests lorsque les échecs nuisant à l'expérience de jeu ont disparu.

— **Priorité élevée :** Cette catégorie représente tous les bogues qui affectent énormément le jeu et qui l'empêchent de s'exécuter de la manière désirée. Par exemple, s'il est impossible de sauter alors que le joueur doit normalement pouvoir le faire, on parle d'une priorité élevée. Ces bogues doivent être réglés rapidement et ne doivent pas se retrouver sur le marché.

— **Priorité critique :** Ces bogues demandent une attention immédiate de tous les développeurs. Il n'est pas question d'échecs nuisant au déroulement du jeu, mais de ceux qui empêchent tout simplement de jouer comme une corruption des données ou si le jeu cesse de fonctionner soudainement. Le travail sur tous les autres éléments s'arrête afin de trouver une solution.

C'est en prenant en compte de ces informations qu'il est possible, pour les testeurs, de bien effectuer leur travail.

2.2 DÉROULEMENT DES TESTS

Levy et Novak (2010) décrivent bien les différentes étapes de production d'un jeu vidéo et les tests qui les accompagnent. Les tests se déroulent dans toutes les étapes dès l'entrée dans la

phase Alpha où le jeu commence à prendre forme. Le nombre de testeurs varie grandement selon le projet, pouvant aller d'environ cinq testeurs pour de petites compagnies à plusieurs centaines pour de plus grosses comme *Nintendo* et *Microsoft* (Starr, 2007).

La phase Alpha du jeu contient le plus de bogues puisque c'est dans celle-ci que les développeurs mettent en place les principaux éléments du gameplay. Les tests sont effectués par les développeurs eux-mêmes ou par des testeurs expérimentés. Puisque c'est à ce moment que les éléments importants du jeu sont implémentés, une grande partie des bogues à priorité élevée s'y trouvent. Toutefois, on se concentre davantage sur la présence des différentes spécifications retrouvées dans le document de game design que sur l'élimination des bogues, expliquant pourquoi la phase Beta s'y attarde davantage.

Dans la phase Beta, le jeu est normalement complet, c'est à dire que l'on peut se rendre du début à la fin et que les atouts graphiques sont présents. C'est dans cette phase que le plus grand nombre de testeurs participent à assurer la qualité du jeu. Les bogues les plus importants devraient normalement être éliminés ou presque. Le travail consiste davantage à éliminer le plus de bogues à priorité moyenne ou faible. Cette étape sert aussi à améliorer le niveau de difficulté du jeu et à le rendre plus divertissant, si nécessaire. Cette phase est souvent séparée en deux parties : « Closed Beta » et « Open Beta ». Le Closed Beta se déroule à l'interne avec les testeurs engagés et le déroulement est très privé. Parfois, certaines entreprises comme *Blizzard Entertainment* permettent à des joueurs importants de leurs franchises de se joindre à cette étape (Foster, 2013). Le Open Beta est souvent utilisé dans les jeux comportant des composantes en ligne comme les MMORPG (*Massive Multiplayer Online Role Playing Game*). Le jeu s'ouvre au public, gratuitement, pour que les joueurs puissent participer à la recherche des bogues. Le jeu *Star Wars : The Old Republic* avait même attiré deux millions de testeurs dans cette phase (Daniel, 2011).

Figure 2.2: Capture d'écran du jeu *The Elder Scrolls 5 : Skyrim*

Normalement, le jeu entre dans la phase Gold lorsqu'il est prêt à sortir sur le marché. Toutefois, le développement n'est pas tout à fait fini. Les consoles de jeu étant désormais constamment connectées sur le web, les entreprises peuvent offrir des « patchs » pour régler des problèmes rencontrés après la sortie. En avril 2013, le jeu *Elder Scrolls 5 : Skyrim* offrait une nouvelle patch aux joueurs alors que le jeu était en vente depuis novembre 2011 (Farokhmanesh, 2013). Les joueurs ont toutefois beaucoup de reproches envers les compagnies sortant des jeux comportant trop de bogues puisqu'ils doivent souvent attendre quelques mois avant de pouvoir jouer sans problème. Pour cette raison, il est encore très important de minimiser le nombre de bogues lors du développement. Pour ce faire, certains outils facilitent le travail des testeurs.

2.3 OUTILS DE TESTS

Pour obtenir de meilleurs résultats dans la recherche des bogues, les entreprises utilisent une organisation propre à chacune. Toutefois, certains outils sont indispensables pour la plupart

d'entre elles.

2.3.1 LE RAPPORT DE BOGUE

Le rapport de bogue est un document texte décrivant, avec le plus de détails possibles, le problème rencontré afin que le développeur en charge puisse reproduire et régler le bogue. Les éléments importants d'un tel rapport dépendent toujours de l'entreprise, mais plusieurs insistent sur les éléments suivants (Levy et Novak, 2010; Watson, 2013; Sloper, 2013) :

— **Titre représentatif :** Le titre doit être clair et démontrer l'impact de ce dernier en un seul coup d'oeil. Ce dernier doit facilement indiquer quel est le problème et où il se situe.

— **Description :** Ce champ explique en détails la nature du bogue. Si possible, il faut indiquer exactement l'endroit où le problème se passe ainsi que tout ce qui s'est passé pour en arriver à ce résultat. Si possible, il faut ajouter les valeurs importantes comme la distance entre un personnage et un objet du décor si cela peut préciser un lieu spécifique. Dans le meilleur des cas, le développeur doit avoir une idée de la solution en lisant la description.

— **Reproduction du bogue :** Cette section doit décrire, étape par étape, comment répéter le bogue. Normalement, le testeur doit s'assurer de la démarche à suivre pour que le problème se reproduise. Le développeur doit, avec cette information, obtenir les même résultats que le testeur et ainsi reproduire le problème.

— **Sévérité :** Il est important de préciser à quel point le problème est sévère. Il est possible d'indiquer le nombre d'occurrences au lieu de la sévérité et laisser le testeur en chef décider de l'importance de celui-ci lors du triage. Un bogue qui arrive à tous les coups est souvent plus sévère qu'un problème se produisant une fois sur dix.

Bettenburg et al. (2008) ont, quant à eux, étudié les éléments importants qui se retrouvent dans un bon rapport de bogue, se concentrant toutefois sur les logiciels traditionnels, donnant des résultats similaires. Ils découvrent que malgré la faible présence des situations de test, à l'instar d'un vidéo représentatif du bogue, dans les rapports, elles représentent l'élément le plus utile pour aider les développeurs à reproduire le problème. L'auteur explique cette faible présence par la difficulté de fournir ces situations de test. L'ajout d'une fonctionnalité permettant d'obtenir des captures d'écran du problème ou de rejouer un vidéo de la séquence pourrait grandement faciliter la tâche des développeurs. La plupart du temps, cette étape est facilitée par la présence des « bug trackers ».

2.3.2 LES BUG TRACKERS

Les « bug trackers » sont des logiciels permettant d'améliorer la gestion des bogues rencontrés lors du développement d'un système (Levy et Novak, 2010). Lorsqu'un testeur rencontre un problème, il se doit d'ajouter le bogue au système en entrant toutes les informations nécessaires, notamment les informations mentionnées dans la section précédente. Ce dernier doit s'assurer qu'il apporte un nouveau problème en vérifiant que ce bogue n'est pas déjà présent dans la base de données. Par la suite, le testeur en chef filtre les nouvelles entrées en s'assurant qu'il s'agit bien de nouveaux bogues et que ces derniers sont clairs et précis pour les développeurs. Les problèmes sont ensuite envoyés aux chefs des départements concernés, ceux-ci redirigeant, à leur tour, les bogues vers les développeurs en charge de les régler. Ces outils permettent aussi de préciser l'état du bogue. Lorsque le testeur entre le bogue dans le système, le bogue est *ouvert*. Il devient *assigné* lorsqu'un développeur s'en occupe et il changera d'état à *fixé* ou *résolu* lorsque le développeur règle le problème. Un testeur vérifie par la suite qu'il a bien disparu et change l'état à *vérifié*. Il existe plusieurs bug trackers, mais plusieurs gros noms de l'industrie comme Activision et EA se tournent vers *DevTrack*

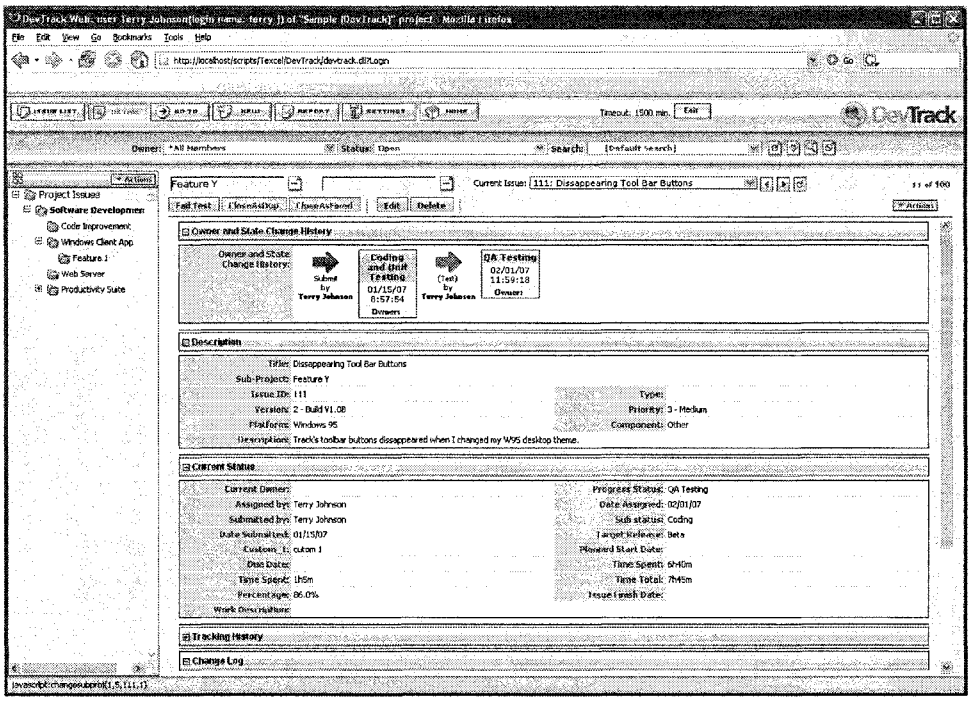


Figure 2.3: Capture d'écran du menu principal de *DevTrack*

(TechExcel, 2010).

2.4 PROBLÈMES RELIÉS AUX TESTS

Malgré le grand nombre de personnes travaillant pour assurer la qualité du jeu, il y a toujours de nombreux bogues qui réussissent à se tailler une place jusqu'au produit final. L'approche par tests utilisée dans les entreprises fait intervenir des tâches répétitives dans lesquelles des scénarios correspondant à tous les cas de figure sont rejoués à chaque modification afin de s'assurer que les éléments déjà en place n'ont pas été altérés et que les nouveaux fonctionnent correctement. Le testeur doit, manuellement, s'assurer que les résultats, que ce soit une séquence d'actions, une modification du pointage ou le changement d'état des personnages, correspondent au design du jeu. Cette méthode, bien qu'acceptée dans l'industrie, pose trois

problèmes. Premièrement, ce qui doit être observé peut parfois être très subtil et difficilement observable par l'être humain. La position du personnage pourrait être décalée ou encore, l'état du personnage pourrait changer sans donner d'indications réelles à l'utilisateur. Ces problèmes, bien qu'inoffensifs à première vue, pourraient amener des problèmes futurs. Par exemple, le bogue de *Dragon Age 2* (Bartel, 2011) mentionné plus haut sur la réduction de la vitesse d'attaque se fait très lentement et peut ne pas être perçu par le testeur sur une courte durée. Deuxièmement, une erreur humaine peut se glisser dans l'interprétation du résultat d'un test. Le testeur peut trouver normal un comportement qui ne devrait pas arriver et ainsi ignorer un bogue potentiel. Troisièmement, le nombre de répétitions du test est limité. En fait, chaque test demande une intervention manuelle qui prend du temps, rendant pratiquement impossible d'observer toutes les situations possibles. Ce point est facilement prouvé par les nombreux bogues observables sur le marché.

C'est pourquoi de nouvelles méthodes doivent être étudiées afin d'améliorer le processus de découverte des bogues. Le runtime monitoring, en particulier, peut permettre d'aider l'être humain à détecter les problèmes qu'il ne remarque pas tout en augmentant l'efficacité de chaque test comme nous le verrons dans la suite de ce mémoire.

CHAPITRE 3

LE RUNTIME MONITORING

Pour résoudre les problèmes mentionnés au chapitre précédent, de nombreuses techniques automatiques ou semi-automatiques d'identification de bogues ont été développées. Plusieurs techniques de vérification ont donc été inventées comme le « theorem proving » (Bonacina, 2010), le « model checking » (Clarke et al., 1999) et les pratiques par tests comme dans le cas des jeux vidéo. Ainsi, le theorem proving consiste à démontrer l'exactitude d'un programme à l'instar d'une preuve mathématique prouvant l'exactitude d'un théorème. C'est une méthode de vérification statique qui n'a pas besoin de l'exécution du système pour accomplir sa tâche. Le model checking examine automatiquement si un système à états finis respecte une spécification donnée. Les tests consistent à retrouver le plus de bogues possibles en exécutant le système à maintes reprises. Ce sont deux techniques de vérification dynamique, c'est-à-dire qu'ils utilisent l'exécution du programme pour fonctionner. Dans le cas du model checking, on calcule les états de l'exécution sans toutefois exécuter le programme alors que les tests requièrent un programme en cours d'exécution. Toutefois, ces techniques, à l'exception des approches par tests, ne permettent pas d'observer ce qui se passe réellement lors de l'exécution, ignorant ainsi les bogues causés par l'émergence. Les pratiques par tests demandent toutefois l'observation constante d'un utilisateur, ce qui amène aussi quelques limitations.

Une technique particulière, le *runtime monitoring*, permet l'exécution d'un système et d'observer lorsque le comportement du système dévie d'une spécification donnée à l'avance. L'efficacité de cette méthode peut être observée à travers des implémentations importantes comme JavaMOP (Meredith et al., 2011), LARVA (Colombo et al., 2009), Tracematches (Bodden et al., 2007) et JLO (Stolz et Bodden, 2006). Toutefois, le runtime monitoring n'a pas encore fait d'entrée majeure dans le domaine des jeux vidéo.

Ce chapitre commencera par fournir les définitions requises pour une bonne compréhension du concept. Ensuite, nous verrons en quoi le runtime monitoring peut s'avérer utile dans le domaine des jeux vidéo et fournirons une explication détaillée du fonctionnement. Par la suite, nous étudierons l'implémentation de la méthode, ce qui demandera aussi des explications sur la représentation des spécifications qui seront écrites en logique.

3.1 DÉFINITION DU MONITORING

Pour expliquer clairement ce qu'est le runtime monitoring, il est important de préciser la signification de quelques mots ou appellations dans ce contexte.

— **Système monitoré :** C'est le système dont l'exécution nous intéresse. Nous tentons de détecter les bogues présents dans ce dernier.

— **Événement :** Un événement représente l'état du système en indiquant ce qui vient de se produire durant l'exécution. Le système doit envoyer des événements pour communiquer avec le moniteur.

— **Trace :** La trace représente l'exécution du système. C'est à partir d'elle que le moniteur peut obtenir le verdict d'une propriété.

— **Spécification :** Une spécification, aussi appelée propriété, décrit le comportement attendu du système et est exprimée en termes des séquences d'événements qu'il doit

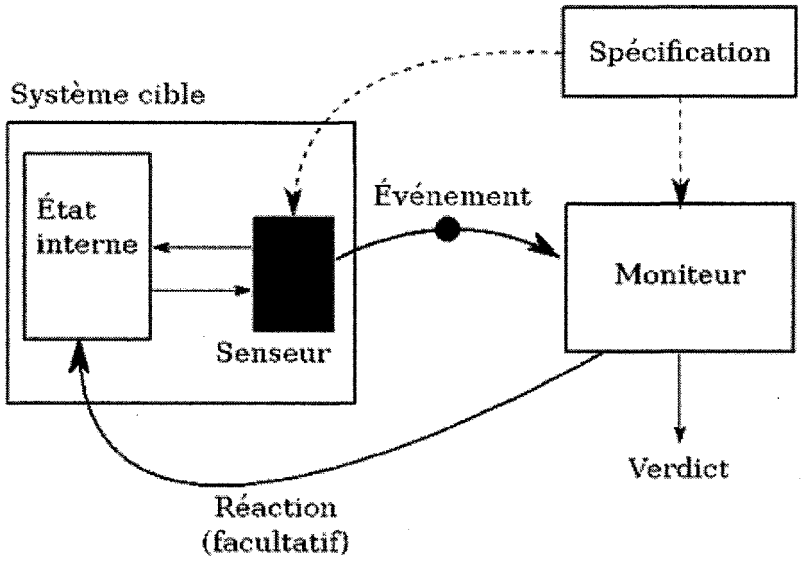


Figure 3.1: Représentation graphique du fonctionnement d'un moniteur

produire.

— **Moniteur :** Leucker et Schallhart (2009) donnent cette définition d'un moniteur : « Un moniteur est un dispositif qui lit une trace finie et qui rapporte un certain verdict. ». C'est l'entité qui va surveiller l'exécution du système. Le moniteur peut soit être directement intégré au système, ou opérer sous la forme d'un programme indépendant.

— **Verdict :** Le verdict indique si la spécification a été violée ou non. Les résultats possibles sont : vrai, faux et non concluant (Leucker et Schallhart, 2009). La signification de chacun de ces résultats est décrite plus loin dans ce chapitre.

Ces termes seront utilisés régulièrement dans ce mémoire puisqu'ils font partie intégrante du runtime monitoring.

Pour que le runtime monitoring fonctionne, il doit y avoir un moyen de communication entre le système scruté et le moniteur. Il faut intégrer une solution permettant d'examiner ce

qui se passe à l'intérieur, c'est-à-dire acquérir l'information qui se trouve dans le système et envoyer ces données au moniteur pour qu'il puisse fournir un verdict (Nusayr et Cook, 2009). La forme que prend ces données varie d'un moniteur à un autre. Par exemple, le moniteur BeepBeep (Hallé et Villemaire, 2009) lit un document XML pour obtenir différents paramètres du message alors que Mayet (Lewis et Whitehead, 2011) accepte les messages utilisant ActiveMQ de Apache. L'instrumentation se fait généralement selon deux méthodes : l'instrumentation manuelle et la programmation orientée aspect.

L'instrumentation manuelle est la façon la plus rudimentaire de s'y prendre et est utilisée pour le moniteur Mayet (Lewis et Whitehead, 2011). Il suffit d'aller insérer, dans le code source, des lignes de code produisant les messages aux endroits appropriés. Par exemple, dans le cas de Lewis et Whitehead (2011), un message est envoyé lorsque le personnage saute. Les auteurs ont repéré l'endroit dans le code où Mario saute et ont simplement ajouté la production du message à la suite des instructions de saut. L'avantage de l'instrumentation manuelle consiste en sa facilité. On ajoute la ligne de notre choix et on instrumente seulement les endroits qui nous intéressent. Le langage du système sous surveillance n'a aucun impact sur la difficulté de l'implémentation puisqu'il n'y a qu'à s'adapter au langage. Par contre, cette solution est de moins en moins efficace plus le système gagne en taille car le nombre d'endroits à modifier devient de plus en plus grand, rendant le processus long et pénible. Voici un exemple d'instrumentation manuelle :

```
Monitor.update("<event><name>cliquerBouton</name><id>Button1</id>
                                                    </event>")
```

Pour éviter ce problème, plusieurs chercheurs se sont tournés vers la programmation orientée aspects (Aspect Oriented Programming ou AOP) (Nusayr et Cook, 2009). Cette méthode permet d'instrumenter le code sans que l'utilisateur ait à écrire lui-même les lignes requises.

L'utilisateur écrit des points d'action (« pointcuts ») qui représentent les endroits où le code doit être inséré, souvent à l'appel d'une méthode. Le système exécutera ainsi les instructions spécifiées dans le pointcut avant ou après l'endroit désigné, au choix de la personne en charge d'ajouter les propriétés. C'est ce que l'on appelle le tissage. Cette façon d'instrumenter est très rapide et évite d'envahir le code source du système à surveiller, ce qui lui donne un avantage sur l'instrumentation manuelle. Par contre, les structures d'AOP sont spécifiques à un langage. Par exemple, AspectJ (Kiczales et al., 2001), utilisé dans les travaux d'Amorim et Havelund (2005), n'est compatible qu'avec Java. Bien qu'efficace, la programmation orientée aspect ne peut pas être utilisée dans un contexte où le moniteur doit pouvoir vérifier l'exécution de différents systèmes aux langages différents. Voici à quoi peut ressembler un pointcut sous AspectJ :

```
aspect PointObserving {
  pointcut changes(Point p): target(p) && call(void Point.set*(int));
  after(Point p): changes(p) {
    Iterator iter = p.observers.iterator();
    while (iter.hasNext() ) {
      ...
    }
}
```

Ce bout de code permet de surveiller tous les appels d'une méthode ayant la signature *void Point.set*(int)*. À chaque fois qu'une méthode semblable sera appelée, la portion de code dans la braquette *after* s'exécutera. Dans ce cas-ci, on itère sur les observeurs dans le point.

Les caractéristiques d'un système peuvent parfois amener à croire qu'aucune de ces deux méthodes n'est idéale. Par exemple, la taille des jeux vidéo de grande envergure ne permet

pas une instrumentation manuelle efficace alors que bien qu'il soit possible d'utiliser la programmation par aspect pour un jeu en particulier, le prochain jeu pourrait être écrit dans un autre langage. Afin d'obtenir un moniteur pouvant détecter les problèmes de plusieurs jeux différents, la limitation du langage ne serait pas appropriée. Ce mémoire décrit, au chapitre 4, une méthode pour contrer ces problèmes.

3.2 JUSTIFICATION

Le runtime monitoring s'utilise principalement comme une alternative lorsque les autres méthodes de vérification ne sont pas adaptées pour un système donné. Leucker et Schallhart (2009) décrivent le runtime monitoring comme étant une technique de vérification légère permettant de complémenter d'autres techniques comme le model checking, qui s'adapte mieux dans les systèmes à états finis, et les tests, qui ne peuvent pas toujours couvrir tous les scénarios possibles. Ils précisent que l'intérêt particulier de cette méthode est qu'elle vérifie l'état du système lors de l'exécution, ce qui pose problème pour d'autres techniques. Leucker et al. continuent en décrivant les applications qui profitent des caractéristiques du runtime monitoring :

— Des techniques comme le model checking et la démonstration automatique de théorèmes regardent directement le système pour vérifier l'exactitude de l'implémentation. Toutefois, il est possible que l'exécution du système diffère de l'implémentation. Le runtime monitoring permet de vérifier l'exécution réelle du système et donc de détecter les cas où les actions de ce dernier ne correspondent pas à une spécification donnée.

— Parfois, l'information requise ne peut être disponible que lors de l'exécution. Par exemple, certaines librairies ne sont pas accompagnées du code source et il est donc impossible de vérifier leur fonctionnement sans les exécuter.

— Le comportement d'une application peut grandement dépendre de son environnement, qui peut ne pas avoir de description précise. Dans ces cas, les autres techniques de vérification ne peuvent qu'émettre des suppositions sur l'environnement. Le runtime monitoring regarde le comportement au moment même de l'interaction avec l'environnement, ce qui permet d'observer le comportement réel du système.

— Parfois, dans les systèmes où la sécurité est importante, il est toujours intéressant de vérifier, à l'aide d'un moniteur, les propriétés qui ont été prouvées statiquement afin de s'assurer de leurs bons fonctionnements. Le runtime monitoring est alors utilisé en tandem avec le theorem proving, le model checking et les tests.

On retrouve en particulier unne combinaison de ces éléments dans les systèmes très dynamiques qui dépendent fortement de l'environnement et qui changent constamment, ce qui les rend imprévisibles. Les jeux vidéo en sont un bon exemple, rendant le runtime monitoring intéressant. La prochaine section se concentrera sur la logique mathématique qui occupe une place importante dans le runtime monitoring.

3.3 LANGAGES FORMELS DE SPÉCIFICATION

Les méthodes formelles permettent de démontrer la validité d'un système selon une certaine spécification écrite à l'aide de la logique mathématique. Cette logique permet de représenter le comportement du programme à l'aide d'opérateurs et connecteurs particuliers et est aussi utilisée dans le domaine de l'intelligence artificielle (Minker, 2000). À travers les années, plusieurs extensions ont été proposées pour permettre de représenter des comportements plus élaborés. Dans ce chapitre, nous étudierons deux formes de logique en particulier : la logique propositionnelle et la logique temporelle linéaire (LTL).

3.3.1 LA LOGIQUE PROPOSITIONNELLE

La logique propositionnelle est une branche de la logique étudiant les façons de joindre ou de modifier des expressions pour en faire des plus complexes (Klement, 2005). Cela permet donc de déduire des faits selon les expressions écrites. Ces expressions doivent être déclaratives et peuvent, en principe, être évaluées à vrai ou faux. Prenons un exemple de la sorte :

1. Si Jean a assez d'argent et que sa voiture est réparée, il ira s'acheter une console de jeu.
2. La voiture de Jean est réparée.
3. Jean n'a pas acheté de console de jeu.

À partir ces trois expressions, il est possible de déduire que Jean n'a pas assez d'argent. Il est par la suite possible de faire le même processus de déduction, mais à l'aide d'une formule. Un énoncé logique est représenté par des formules comportant des expressions et des opérateurs logiques. Lorsqu'une expression n'est pas décomposable, on dit qu'elle est atomique (Huth et Ryan, 2004). Voici trois expressions atomiques :

— p : Jean a payé les réparations de sa voiture.
— q : La voiture de Jean est réparée.
— r : Le mécanicien remet la voiture de Jean à la mauvaise personne.
— s : Jean conduit sa voiture.

Il est ensuite possible d'écrire une formule représentant une situation. Dans ce cas-ci, on peut dire que Jean ne peut conduire sa voiture que s'il a payé le coût des réparations, que sa voiture est réparée et que le mécanicien n'a pas remis la voiture à la mauvaise personne. Pour représenter le tout par une formule, il faut utiliser les opérateurs logiques suivants (Huth et

Ryan, 2004) :

- $\neg$: La négation inverse la valeur d'une expression. L'expression $\neg q$ indique que la voiture de Jean n'est pas réparée.
- $\wedge$: La conjonction est l'équivalent de « et ». L'expression $p \wedge q$ indique que Jean a payé les réparations de la voiture et que sa voiture est réparée.
- $\vee$: La disjonction est l'équivalent de « ou ». L'expression $s \vee r$ indique que la voiture de Jean conduit sa voiture ou que le mécanicien l'a remise à la mauvaise personne.
- $\rightarrow$: Ce symbole représente l'implication. Par exemple, $s \rightarrow q$ précise que si Jean conduit sa voiture, alors cela veut dire que sa voiture est réparée. Une expression contenant une implication est aussi évaluée à vrai lorsque l'élément précédent l'opérateur est évaluée à faux. En reprenant le même exemple, l'expression est aussi vraie lorsque Jean ne conduit pas sa voiture, peu importe qu'elle soit réparée ou non.

En utilisant ces opérateurs, il est possible d'écrire la formule suivante :

$$s \rightarrow (p \wedge q \wedge \neg r)$$

Ainsi, il est possible de lire que si Jean conduit sa voiture, cela signifie qu'elle est réparée, qu'il a payé les réparations et que le mécanicien n'a pas remis la voiture à une autre personne. Si on regarde le tout selon un point de vue de spécification d'un système, il serait possible d'évaluer cette formule à faux si s est vraie et que l'une des trois expressions atomique ne s'applique pas. Par contre, la logique propositionnelle est limitée. Elle ne permet pas d'exprimer des énoncés sur plusieurs moments dans le temps, l'empêchant de servir efficacement pour le runtime monitoring.

3.3.2 LA LOGIQUE DU PREMIER ORDRE

La logique du premier ordre se distingue de la logique propositionnelle par son utilisation de prédicats et de quantificateurs. La logique précédente est efficace lorque l'on désire utiliser des composants de phrases comme « et », « ou », « si...alors » et « non », mais ces limitations nous empêchent d'écrire un énoncé de ce genre (Huth et Ryan, 2004) :

Tous les lutteurs sont plus forts qu'au moins un informaticien.

Cette phrase insiste sur trois éléments en particuliers : être un lutteur, être un informaticien et être plus fort que quelqu'un d'autre. Pour ce faire, on utilise les prédicats. On peut donc écrire $L(shawn)$ pour indiquer que Shawn est un lutteur ou $I(richard)$ pour dire que Richard est un informaticien. Il est aussi possible d'écrire des prédicats d'arité 2, indiquant qu'il y a deux paramètres, de la même façon : $F(shawn, richard)$ signifie que Shawn est plus fort que Richard. Toutefois, pour l'instant, il n'est pas encore possible de dire que tous les lutteurs sont plus forts qu'un informaticien. Il faut donc ajouter les quantificateurs de premier ordre :

- $\forall x$: Ce quantificateur représente « Pour tout« . Cela signifie que pour tout x, ce qui suit est vrai. Par exemple, l'expression $\forall x : (x > 3)$ précise que tous les x doivent être plus grand que 3.
- $\exists x$: Ce quantificateur représente « Il existe ». Cela signifie qu'il existe au moins un x où ce qui suit est vrai. Par exemple, l'expression $\exists x : (x > 3)$ précise qu'il doit y avoir au moins un x plus grand que 3.

Il est désormais possible, à l'aide des prédicats et des quantificateurs, d'écrire la première phrase sous forme logique suivante :

$$\forall x \exists y : (L(x) \wedge I(y) \rightarrow F(x,y))$$

Cette formule indique que pour chaque lutteur, il existe au moins un informaticien qui sera moins fort que lui. Toutefois, le problème de la représentation dans le temps apportée par la logique propositionnelle est encore présent, ce qui pose une limitation pour le runtime monitoring.

3.3.3 LA LOGIQUE TEMPORELLE LINÉAIRE

Dans le domaine de la vérification automatique, que ce soit le model checking ou le runtime monitoring, la logique propositionnelle et du premier ordre ne permettent pas d'obtenir de résultats utiles. Elles nous permettent d'énoncer des faits sur l'état d'un système à un moment donné, mais ne prennent pas en compte le temps. Par exemple, prenons un cas où à chaque fois que l'on clique sur un bouton, on ouvre une fenêtre. Il est possible de dégager deux événements importants de cette situation : cliquer sur un bouton et ouvrir une fenêtre. Il serait tentant d'écrire :

$$\text{cliquerBouton} \rightarrow \text{ouvrirFenêtre}$$

Pourtant, deux problèmes se présentent. En premier lieu, au moment où on clique sur le bouton, la fenêtre n'est pas ouverte, bien qu'elle le sera bientôt. La formule plus haut voit qu'il y a un *cliquerBouton*, mais pas d'*ouvrirFenêtre*, ce qui évaluerait la formule à faux malgré que la fenêtre soit sur le point de s'ouvrir. On ne voit qu'un seul instant et non pas l'exécution du système. En deuxième lieu, même si la formule s'évalue à vrai ou faux, rien indique que ce sera le cas durant toute l'exécution : il y a des chances que l'utilisateur clique sur d'autres

boutons. Il faut que la formule soit valable durant toute l'exécution du système, ce qui n'est assurément pas le cas pour l'instant. La logique temporelle linéaire (LTL) permet justement d'exprimer ce genre d'énoncés en introduisant des connecteurs temporels (Huth et Ryan, 2004).

— **G** (Globalement) : L'expression doit être vraie pour toute l'exécution du système.
— **X** (Prochain état) : Indique ce que doit être le prochain état.
— **F** (Dans le futur) : Indique qu'une expression doit être vraie au moins une fois durant l'exécution.
— **U** (Jusqu'à) : Indique que l'expression précédant le « **U** » doit être vraie jusqu'à ce que l'expression suivant le « **U** » soit vraie.

Afin d'évaluer des formules LTL, il faut utiliser les règles récursives qui suivent lorsque l'on a une trace σ donnée représentant une suite d'événements donnée par l'exécution d'un système. La notation $\models$ indique qu'une trace satisfait une expression φ. On dit donc que σ est un modèle de φ. On utilise σ^2 pour représenter le reste de la trace sans inclure le premier élément. Lorsque l'on évalue si une trace est un modèle d'une expression, on compare l'expression avec le premier élément de la trace.

1. $\sigma \models a \equiv \sigma_1 = a$
2. $\sigma \models \neg\varphi \equiv \sigma \not\models \varphi$
3. $\sigma \models \varphi \wedge \psi \equiv \sigma \models \varphi \ \textit{et} \ \sigma \models \psi$
4. $\sigma \models \varphi \vee \psi \equiv \sigma \models \varphi \ \textit{ou} \ \sigma \models \psi$
5. $\sigma \models \varphi \rightarrow \psi \equiv \sigma \not\models \varphi \ \textit{ou} \ \sigma \models \psi$
6. $\sigma \models \mathbf{G}\varphi \equiv \sigma \models \varphi \ \textit{et} \ \sigma^2 \models \mathbf{G}\varphi$
7. $\sigma \models \mathbf{F}\varphi \equiv \sigma \models \varphi \ \textit{ou} \ \sigma^2 \models \mathbf{F}\varphi$
8. $\sigma \models \mathbf{X}\varphi \equiv \sigma^2 \models \varphi$
9. $\sigma \models \varphi\mathbf{U}\psi \equiv \sigma \models \psi \ \textit{ou} \ (\sigma \models \varphi \ \textit{et} \ \sigma^2 \models \varphi\mathbf{U}\psi)$

Prenons, par exemple, la règle numéro 8 en utilisant la propriété $\sigma \models \mathbf{F}a$ et la trace *cabb*. On vérifie tout d'abord la règle sur le premier élément, qui nous indique que l'élément suivant doit être *a*. Afin de vérifier cette règle, on évalue par la suite si *a*, le deuxième élément de la trace, est valide. Dans ce cas-ci, $abb \models a$ puisque le premier élément est *a*.

Reprenons l'exemple du début en prenant en compte la logique LTL. Il faut qu'une fenêtre s'ouvre à l'état suivant le clic sur le bouton, ce qui est représenté par le connecteur **X**. Par la suite, on veut appliquer cette formule durant toute l'exécution afin que l'on surveille chaque clic, ce qui sera représenté par le connecteur G. On obtiendrait la formule suivante :

$$\mathbf{G}(\text{cliquerBouton} \rightarrow \mathbf{X}\ \text{ouvrirFenêtre})$$

La formule dit : « globalement, l'événement cliquerBouton implique que le prochain événement doit être ouvrirFenêtre ». Le verdict de cette formule sera faux dès qu'elle sera violée au moins une fois. Elle ne sera aussi jamais évaluée à vraie puisqu'il sera toujours possible que dans le futur, cette propriété ne soit pas respectée. Elle sera donc évaluée à « non-concluante ».

Cette base en logique permettra de mieux expliquer les différentes propriétés à surveiller dans les jeux vidéo qui seront décrites plus loin dans ce mémoire. La prochaine section précisera davantage l'utilité de la logique dans le runtime monitoring en expliquant le fonctionnement de la technique.

3.3.4 SPÉCIFICATION

Une spécification représente le comportement attendu du système. Pour être efficace, il faut donc avoir une bonne connaissance du système sous surveillance afin de bien énumérer les comportements attendus. Par exemple, l'itérateur Java nécessite que la méthode « *hasNext()* »

soit appelée et qu'elle retourne vrai avant un appel de la méthode « *next()* ». Cette propriété permet donc d'empêcher un utilisateur d'itérer à l'extérieur d'une Collection Java. Il y a différentes façons d'écrire une spécification et cela dépend entièrement du moniteur.

D'autres méthodes se tournent vers les automates (Meredith et al., 2011; Falcone et al., 2008). Les automates ainsi créés servent à représenter l'exécution désirée d'un système. En reprenant l'exemple de l'itérateur Java, la méthode *hasNext()* retournant vrai nous amènerait vers un état permettant d'utiliser la méthode *next()* alors que dans le cas une réponse négative, nous nous dirigerions vers un état où *next()* représenterait une erreur.

Certains moniteurs, comme LARVA (Colombo et al., 2009) demandent d'écrire les spécifications sous une forme propre au moniteur. LARVA utilise, par exemple, le LARVA script. Il est aussi possible utiliser des engins de règles à l'instar de Drools, comme c'est le cas de Mayet (Lewis et Whitehead, 2011). Ces différentes façons de faire sont efficaces, mais pas utilisées régulièrement dans le domaine du runtime monitoring puisque que chaque chercheur décide de créer sa propre méthode de spécification. L'apprentissage de chaque syntaxe n'est pas une option et il est préférable de se tourner vers des méthodes mieux établies dans la communauté.

D'autres méthodes utilisent la logique, principalement LTL (Basin et al., 2010; Stolz et Bodden, 2006; Bauer et al., 2011) décrite plus tôt dans ce mémoire. On représente une propriété à surveiller en utilisant les opérateurs de logique et les expressions appropriés. Voici à quoi pourrait ressembler une formule LTL simple si on désire lire un fichier :

$$\mathbf{G}(\neg \text{readFile } \mathbf{U} \text{ openFile})$$

Dans cet exemple, la spécification précise qu'avant de lire le fichier, il faut ouvrir le fichier. La logique permet de représenter un grande partie des propriétés d'un système, rendant son

utilisation intéressante. De plus, elle est largement utilisée sous plusieurs formes dans différents domaines comme le model checking (Clarke et al., 1999). Toutefois, certaines spécifications sont considérées non-monitorables (Leucker et Schallhart, 2009). Par exemple, une propriété indiquant que chaque requête doit être reconnue à un moment donné ne permettrait pas d'offrir un verdict intéressant. Il est toujours possible que plus tard dans l'exécution, on observe une reconnaissance, alors on ne peut jamais dire que la propriété est fausse, mais seulement « non-concluante », ce qui enlève beaucoup d'intérêt à la propriété.

3.4 FONCTIONNEMENT DU RUNTIME MONITORING

En détail, chaque moniteur fonctionne différemment. Par contre, ils partagent tous le même principe. Lorsque l'instrumentation et les propriétés sont mises en place, il faut que les messages sur l'état puissent atteindre le moniteur. Pour ce faire, il est possible d'intégrer directement le moniteur à l'intérieur du système sous surveillance ou de complètement séparer les deux entités. La deuxième solution est souvent considérée la meilleure. Lewis et Whitehead (2011) mentionnent deux raisons. Premièrement, cela permet d'assurer une indépendance du langage. En séparant le moniteur et le système, il est possible d'utiliser deux langages différents en les faisant communiquer par l'envoi de messages. Ainsi, le moniteur peut être utilisé sans problème pour une autre application écrite à l'aide d'un autre langage. Deuxièmement, cela permet d'offrir une indépendance d'exécution. Bien que l'on essaie toujours d'implémenter un moniteur le plus légèrement possible, ils peuvent entraîner un certain coût de performance. En séparant les exécutions, on laisse le système fonctionner à pleine vitesse, ce qui est particulièrement utile pour les gros systèmes, comme les jeux vidéo.

La figure 3.1 représente graphiquement le fonctionnement du runtime monitoring. Le senseur examine l'exécution et envoie des messages au moniteur pour l'informer du déroulement.

Ces messages représentent souvent des événements qui se sont produits à l'intérieur du système (d'Amorim et Havelund, 2005). Par exemple, cliquer sur un bouton représenterait l'événement « CliquerBouton ». Il est aussi possible d'ajouter des paramètres pour augmenter la précision, comme le nom du bouton en question, si le moniteur le permet. Le moniteur traite ces événements et donne un verdict selon la trace obtenue jusqu'à maintenant. Certains moniteurs, comme Mayet (Lewis et Whitehead, 2011) réagissent lorsqu'une propriété est violée et tentent de corriger la situation. Par exemple, dans le cas de Mayet, si Mario saute trop haut, il sera redescendu le plus rapidement possible. Toutefois, cette façon de faire est encore plus intrusive, ce qui n'est pas préférable dans certains cas.

Le moniteur retournera un verdict qui variera entre vrai, faux et *inconclusive* (Leucker et Schallhart, 2009). Prenons la trace *qpr* et la formule LTL suivante :

$$\mathbf{G}(p \rightarrow \mathbf{X}q)$$

Premièrement, puisque le premier événement de la trace est q, la formule est satisfaite dû à la résolution de l'implication. Le deuxième événement, p correspond à la première partie de l'implication. Il faut donc que le prochain événement, représenté par le connecteur X, soit l'événement q. Ce n'est pas le cas, donc la formule est évaluée à faux. La figure 3.2 représente le travail effectué par le moniteur.

Si la trace correspond plutôt à *qpq*, le moniteur évalue plutôt la partie *pq* à vrai. Le verdict final de la trace est donc « non-concluant » puisqu'il est toujours possible que l'exécution continue et que d'autres événements s'ajoutent à la trace et violent la formule.

Il est préférable de conserver la trace de l'exécution par la suite pour pouvoir s'y référer. Ainsi, il sera possible de voir à quel moment le problème a été rencontré pour permettre aux

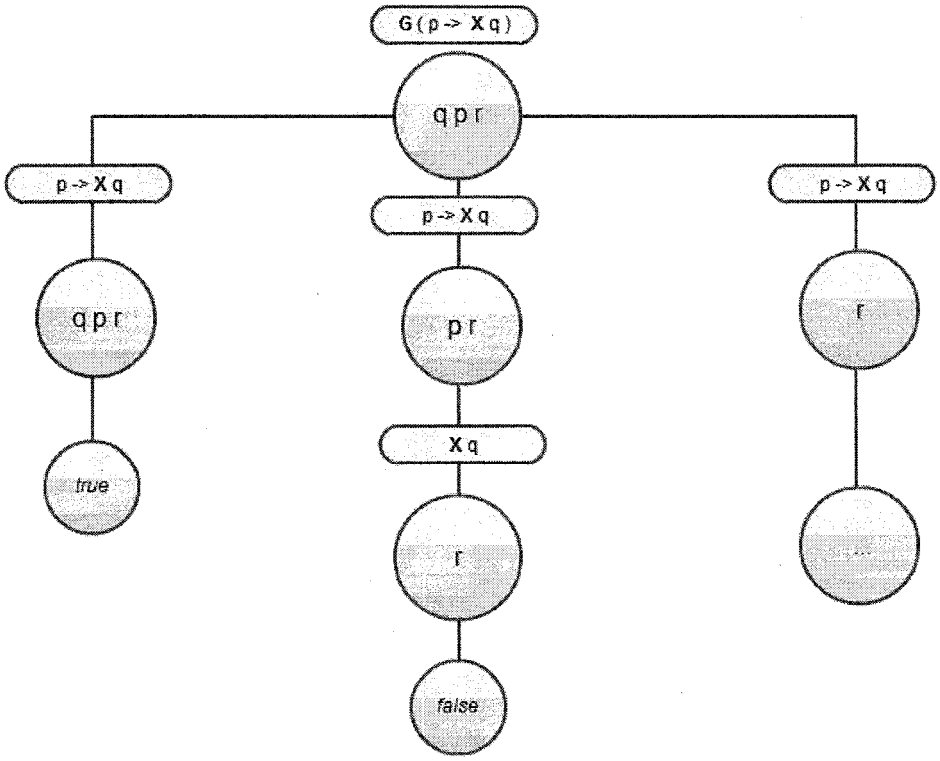


Figure 3.2: Arbre représentant l'évaluation d'une formule LTL sur une trace donnée

développeurs de retrouver la cause de l'échec dans le code. Le prochain chapitre se concentre sur l'application du runtime monitoring avec les jeux vidéo et montre des solutions pour résoudre certains problèmes de la technique appliquée dans ce domaine.

CHAPITRE 4

EXPÉRIMENTATIONS

Les jeux vidéo possèdent plusieurs particularités de l'article de Leucker et Schallhart (2009), ce qui présente le runtime monitoring comme étant une option intéressante. C'est la seule technique des méthodes formelles qui semble adaptée à ce domaine et cela permettrait d'éviter certaines limitations que les tests manuels amènent : la difficulté d'observer certains bogues, l'erreur d'interprétation qu'un humain peut faire et le temps limité ne permettant pas de tester toutes les traces possibles dans un jeu vidéo. Toutefois, les jeux vidéo ont aussi quelques contraintes comme la taille, le langage utilisé et l'importance de n'avoir aucun ralentissement à l'exécution. Il faut donc s'assurer que le runtime monitoring peut satisfaire ces contraintes. Ce chapitre commencera par la description du moniteur choisi pour les expérimentations et des jeux qui serviront de preuves de concept. Lorsque ce sera fait, la nouvelle méthode d'instrumentation permettant d'éviter les limitations imposées par le domaine sera amenée. De nombreux tests ont été effectués pour garantir le bon fonctionnement et l'intérêt de la technique. Ces tests et leurs résultats seront rassemblés au même endroit afin de permettre une interprétation adéquate.

4.1 LE MONITEUR BEEPBEEP

Le moniteur de Hallé et Villemaire (2009), BeepBeep, a été sélectionné pour prouver la faisabilité de l'application du runtime monitoring dans les jeux vidéo. Celui-ci est écrit en Java et accepte des messages sous forme de document XML, ce qui rend facile l'envoi et la réception d'événements. De plus, les propriétés sont écrites à l'aide de LTL-FO+, une extension de LTL, permettant d'accéder aux différents paramètres d'un message à l'aide de XPath, un langage de requêtes pour sélectionner des noeuds dans un document XML. Puisque chaque partie du fonctionnement utilise des syntaxes connues ou, dans le cas de LTL-FO+, une syntaxe basée sur un langage logique utilisé abondamment dans le runtime monitoring, BeepBeep rend possible la surveillance d'application sans trop d'apprentissage, contrairement à d'autres moniteurs créant leurs propres façons de faire (Colombo et al., 2009). Il faut toutefois bien connaître chaque élément du moniteur pour travailler efficacement.

4.1.1 XML ET XPATH

BeepBeep permet l'envoi de messages sous format XML. Chaque message envoyé correspond à un instantané de l'état du système. Cette méthode permet de placer des valeurs comme paramètres et d'y accéder facilement lors du traitement à l'aide de XPath. Un autre avantage est qu'il est toujours possible d'ajouter un paramètre selon nos besoins puisque XML permet d'écrire n'importe quelle balise. Un message simple ressemble à ceci :

```
<event>
  <name>clickButton</name>
  <buttonPressed>Button1</buttonPressed>
</event>
```

Dans ce cas-ci, le message indique qu'il y a un événement du nom de « clickButton » qui concerne l'élément précis « Button1 ». La présence de plusieurs paramètres permet d'écrire des contraintes plus spécifiques comme : « globalement, si on reçoit un événement « clickButton » concernant « Button1 », le prochain événement devra être « ouvrirMenu » ». Cette propriété concerne un seul bouton, ce qui laisse la place à d'autres propriétés concernant d'autres boutons. Cette forme de messages permet aussi d'obtenir l'état de plusieurs éléments différents à chaque pas de temps, ce qui est abordé plus loin dans ce chapitre lors de l'instrumentation spécifique aux jeux vidéo.

Pour obtenir les informations désirées d'un fichier XML, on utilise XPath (Berglund et al., 2010). XPath est un langage de requête servant à sélectionner des noeuds d'un document XML. Ce langage est basé sur la représentation en arbre d'un document XML et permet de naviguer à l'intérieur de ce dernier en spécifiant certains critères. Pour le document XML écrit plus haut, il serait possible d'écrire */event/name* pour obtenir l'élément si situant à l'intérieur de la balise *name*.

4.1.2 ÉCRITURE DES PROPRIÉTÉS

Comme mentionné précédemment, BeepBeep utilise LTL-FO+ pour écrire les propriétés à surveiller, qui se trouve à être une extension de LTL. Il y a deux changements importants comparativement à LTL. Premièrement, on ajoute un élément XPath permettant de trouver, à l'intérieur du document XML reçu, le paramètre à vérifier. L'élément « /event/buttonPressed », si utilisé dans l'exemple plus haut, ressortirait la valeur « Button1 ». Deuxièmement, il y a l'ajout de la logique de premier ordre qui amène les quantificateurs $\forall$ et $\exists$. Cette logique de premier ordre sert principalement lorsqu'il y a plusieurs entités semblables. Lorsqu'on envoie l'état de plusieurs entités dans un système, il y a de bonnes chances que plusieurs d'entre elles

```
<message>
  <characters>
    <character>
      <isalive>true</isalive>
      <id>1</id>
    </character>
    <character>
      <isalive>true</isalive>
      <id>1</id>
    </character>
    <character>
      <isalive>false</isalive>
      <id>1</id>
    </character>
  </characters>
</message>
```

Figure 4.1: Exemple d'un message XML produit par un jeu

soient semblables et que l'on doive appliquer une règle pour chacune. Prenons par exemple une des règles implémentées pour un des jeux étudiés dans ce mémoire où il y a plusieurs personnages à l'écran :

$$\mathbf{G}(\forall x \in /message/characters/character/isalive : (x = true))$$

La formule signifie « globalement, pour toute valeur retrouvée dans la balise *isalive*, cette valeur doit être égale à *true* ». La première chose à remarquer est la présence de XPath qui désigne l'endroit où trouver la valeur qui nous intéresse. La logique de premier ordre permet d'indiquer que l'expression *(x = true)* doit être vraie à toutes les fois que l'on rencontre l'élément dans le document XML. Prenons le XML suivant :

Il est important de regarder ce que XPath va chercher. La formule indique que l'on va regarder les éléments trouvés dans une balise *isalive* qui se trouve elle-même à l'intérieur d'une série

de balises (/textit/message/characters/character, ce que nous avons trois fois dans le message précédent. En recevant ce message, le verdict de la formule est *false* puisqu'il y a un endroit où la valeur retrouvée n'est pas *true*. Par contre, en remplaçant $\forall$ par $\exists$ dans la formule, l'expression est satisfaite puisqu'il y a au moins un endroit où l'on retrouve *true*. Le verdict du moniteur demeure toutefois *inconclusive* puisqu'il y a la présence du connecteur temporel G et qu'il y a toujours la possibilité de recevoir d'autres messages.

4.2 MÉTHODE D'INSTRUMENTATION DANS LES JEUX VIDÉO

Pour les expérimentations de ce mémoire, une nouvelle approche à l'instrumentation a été élaborée. Suite à une instrumentation manuelle sur deux jeux, le problème du temps requis pour l'apprentissage du code et de l'instrumentation devenait plus important. Toutefois, il n'était pas envisageable d'utiliser la programmation orientée aspects (Stolz et Bodden, 2006) puisque les deux jeux utilisent des langages de programmation différents, Java et C#, et que le travail allait continuer en utilisant des jeux écrits en C++. La technique élaborée utilise un principe important dans les jeux vidéo, la boucle de jeu, et un moteur de gabarit (*templating engine*).

4.2.1 LA BOUCLE DE JEU

Un jeu vidéo est la représentation d'un monde qui, même si l'utilisateur ne fait rien, change au cours du temps. Dans un logiciel traditionnel, le système attend une entrée de l'utilisateur avant d'agir. Du côté du jeu, l'intelligence artificielle fait évoluer les ennemis alors que le joueur n'interagit pas avec le système. Ainsi, le jeu doit pouvoir mettre à jour son état, que ce soit pour gérer l'intelligence artificielle ou les entrées du joueur. C'est ce qu'on appelle la boucle de jeu (Foltz, 2011). Normalement, un jeu doit se rafraîchir un certain nombre

de fois par seconde afin d'être fluide aux yeux du joueur. Ce nombre représente le taux de rafraîchissement ou *frame rate*. Le taux de rafraîchissement (*frame rate*) varie d'un jeu à l'autre, mais il est courant de voir des nombres variant de 30 à 60. La présence de la boucle de jeu indique qu'il est possible d'obtenir de l'information sur l'état du jeu plusieurs fois par seconde. De plus, il est aussi possible d'interroger les différentes entités du jeu à l'intérieur de cette boucle afin d'obtenir de l'information sur les points de vie d'un ennemi, la position de chaque entité ou encore, quelle action est entreprise par l'unité. On peut ainsi questionner l'état du jeu en ne modifiant qu'un endroit : la boucle de jeu. Cette façon de faire permet d'éviter de rechercher et modifier plusieurs endroits dans le code, rendant le travail beaucoup moins pénible. Cette méthode est aussi moins intrusive que l'instrumentation manuelle traditionnelle.

4.2.2 L'UTILISATION D'UN TEMPLATING ENGINE

Au lieu d'envoyer des messages décrivant les différents événements du système (Lewis et Whitehead, 2010), cette méthode requiert de prendre des « photos » de l'état du système à chaque tour de boucle. Elle demande de structurer correctement l'information, ce qu'un *templating engine* permet de faire. Le choix s'est arrêté sur un engin écrit en C++ : *cpptempl* (Ginstrom, 2010). Voici un extrait de code pouvant produire un HTML :

Cet exemple commence tout d'abord en indiquant le template dans une wstring. Ce qui sera écrit entre les % permet de faire une boucle qui écrira autant d'endroits que ce qui sera fourni par le programmeur. Les variables sont représentées en utilisant le symbole $. Par la suite, on ajoute les éléments désirés, des endroits pour cet exemple, en les ajoutant à l'intérieur d'une *data_ list*. Lorsque cette liste est complète, on l'envoie dans une *data_ map* représentant tout nos éléments. Ces lignes de code produiront le message suivant :

L'engin permet d'ajouter plusieurs éléments dans un *template*. Au lieu d'un HTML, il est aussi

```
wstring text = L"<h3>Locations</h3>\n<ul>\n"
  L"{% for place in places %}"
  L"<li>{$place}</li>\n"
  L"{% endfor %}"
  L"</ul>" ;
// On crée la liste d'endroits
cpptempl::data_list places;
places.push_back(cpptempl::make_data(L"Montreal"));
places.push_back(cpptempl::make_data(L"Saguenay"));
// On ajoute dans une data_map
cpptempl::data_map data;
data[L"places"] = cpptempl::make_data(places);
// Il faut transformer le tout en wstring
wstring result = cpptempl::parse(text, data) ;
```

Figure 4.2: Exemple de l'utilisation de *cpptempl*

```
<h3>Locations</h3>
<ul>
  <li>Montreal</li>
  <li>Saguenay</li>
</ul>
```

Figure 4.3: XML produit par le code de la figure 4.2.

facile d'ajouter différents éléments dans une structure XML, HTML étant très ressemblant à XML. Afin de permettre le runtime monitoring dans les jeux vidéo, on ajoute une structure semblable à la figure 4.2 dans la boucle de jeu afin de recueillir l'information sur les différentes entités du jeu. L'utilisation d'un moteur n'est pas obligatoire puisqu'il est possible de produire le XML destiné à BeepBeep manuellement, ce qui prend toutefois plus de temps. Plus loin dans ce chapitre, il sera possible de voir l'application de du moteur dans les jeux vidéo.

4.2.3 COMMUNICATION ENTRE BEEPBEEP ET LE JEU

En utilisant la boucle de jeu et un *templating engine*, il est possible de produire le document XML requis pour que BeepBeep puisse monitorer les différentes propriétés. Il faut ensuite établir une connexion entre le moniteur et le système sous surveillance. Comme vu précédemment, il est préférable de séparer le moniteur et le système pour permettre une indépendance des langages et de l'exécution. Pour ce faire, la technique utilise un *pipe* dans lequel le jeu envoie l'information sous la forme d'un XML. BeepBeep, de son côté, prend l'information se retrouvant dans ce *pipe* en le lisant constamment pour obtenir les dernières données envoyées par le jeu. Le moniteur traite ensuite le XML et compare la trace avec les différentes contraintes écrites par le développeur. On crée ainsi un lien entre le jeu et le moniteur tout en les gardant complètement indépendants. Cette méthode est utilisée dans deux des quatre jeux étudiés dans ce mémoire mentionnés dans la section 4.3.

4.3 JEUX VIDÉO ÉTUDIÉS

Les jeux sélectionnés pour ce mémoire représentent quatre types de jeux différents afin d'étudier la faisabilité du runtime monitoring dans plusieurs milieux différents : *Infinite Mario Bros*, un jeu de plateforme, *The Timebuilders : Pyramid Rising 2*, un jeu où l'on gère des

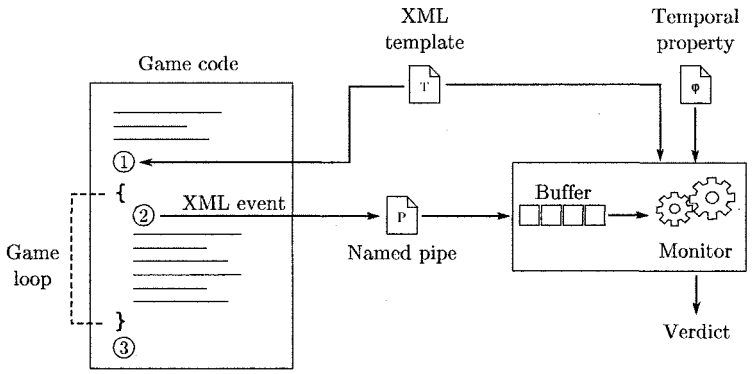


Figure 4.4: Un sommaire de l'architecture du runtime monitoring. Les numéros 1-3 indiquent les trois endroits où des lignes de code doivent être insérés pour instrumenter le programme. 1) Instanciation du template ; 2) Création du template dans la boucle de jeu ; 3) Destruction du template

ressources pour accomplir des objectifs variés, *Pingus*, un jeu de puzzle où l'on doit escorter des pingouins vers la sortie et *Bos Wars*, un jeu de stratégie en temps réel où il faut éliminer les ennemis. Cela permet d'obtenir une meilleure couverture et donner une meilleure crédibilité à l'étude. Ce mémoire a toutefois été limité dans sa sélection pour des jeux dits *open source* à l'exception d'un seul où une collaboration avec une entreprise a permis de travailler sur un jeu commercial. Afin de pouvoir vérifier différentes propriétés, des bogues ont été ajoutés dans chacun, sauf dans le dernier qui utilise de vrais bogues qui se retrouvent dans la version du jeu.

4.3.1 INFINITE MARIO BROS

Infinite Mario Bros (Persson, 2006) est une ré-implémentation du populaire jeu *Super Mario Bros* en utilisant le langage Java. C'est un jeu de plateforme où le joueur contrôle Mario, dont le but est de passer les différents niveaux pour se rendre jusqu'à la fin. Par contre, cette version du jeu continue à l'infini, permettant seulement au joueur de passer des niveaux sans arrêter. Pour avancer, le joueur doit faire sauter Mario tout au long du parcours, éliminant ses ennemis en sautant sur leur tête. Il y a aussi la possibilité de ramasser des pièces pour gagner d'autres

```
<action>
  <name>Stomp</name>
  <id>7</id>
  <iswinged>true</iswinged>
</action>
```

Figure 4.5: Exemple d'un message XML produit par *Infinite Mario Bros*.

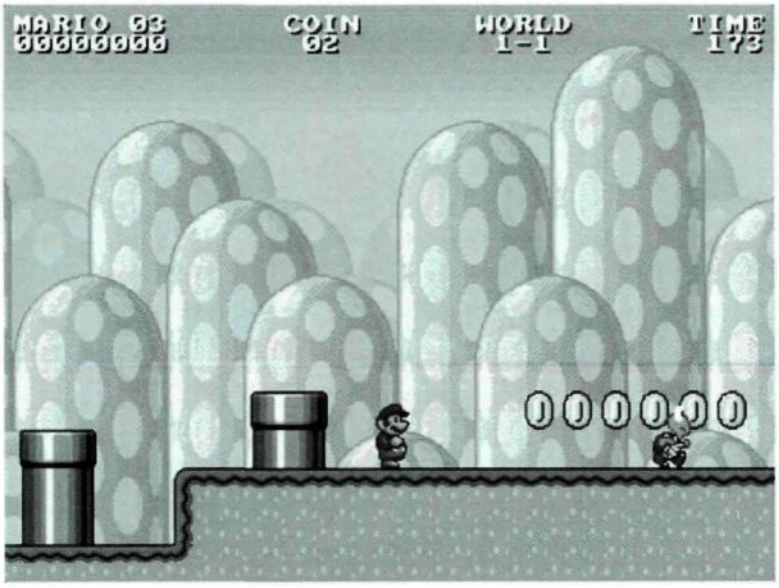


Figure 4.6: Capture d'écran du jeu *Infinite Mario Bros*.

vies, de grandir en mangeant un champignon, de lancer des boules de feu en ramassant une fleur et de ramasser des carapaces pour lancer sur des ennemis. *Infinite Mario Bros* est le premier jeu où le runtime monitoring a été intégré. Contrairement aux autres jeux étudiés par la suite, le moniteur a été directement ajouté et compilé au jeu. De plus, l'instrumentation s'est faite manuellement, le but étant de voir s'il était possible d'observer des propriétés d'un jeu vidéo à l'aide de la logique temporelle linéaire. Voici un exemple de message produit :

Cet événement indique qu'une action nommée « Stomp » a été effectuée sur l'ennemi portant l'identifiant 7 et que ce dernier possède des ailes.

Des bogues ont été ajoutés manuellement afin de pouvoir observer des échecs puisque le jeu

ne possède pas de bogues connus. Ces bogues ont été ajoutés en enlevant ou en modifiant certaines lignes de code. Plusieurs propriétés ont été implémentées afin de voir combien de contraintes pouvaient être surveillées lors d'une exécution, mais nous nous attardons qu'aux plus importantes :

Propriété de monitoring 1

$$\boldsymbol{G}(\forall h \in /action/jumpHeight : (h < 20))$$

Cette formule signifie « globalement, toute valeur h de *jumpHeight* doit être inférieure à 20 unités de hauteur ». Ce nombre représente la limite de l'écran. Normalement, dans le jeu, il est possible de dépasser l'écran, mais cette propriété permettait de comparer des valeurs pour obtenir un verdict.

Propriété de monitoring 2

$$\boldsymbol{G}((\forall x_1 \in /action/name : (x_1 = EnemyFireballDeath)) \rightarrow \boldsymbol{X}(\forall x_2 \in /action/name : (x_2 = FireballDisappear)))$$

Cette formule se traduit par « globalement, toute action *EnemyFireballDeath* doit être immédiatement suivie par une action *FireballDisappear* ». Dans le jeu, il faut qu'une boule de feu lancée par Mario disparaisse après avoir tué un ennemi afin de s'assurer qu'elle ne passe pas au travers d'un monstre pour toucher d'autres unités.

Propriété de monitoring 3

$$\boldsymbol{G}((\forall x_1 \in /action[name = Stomp]/iswinged : (x_1 = true)) \rightarrow (\forall x_2 \in /action/id : \boldsymbol{X}(\forall x_3 \in /action[id = \$x_2]/name : \neg(x_3 = EnemyDead))))$$

Cette formule, plus complexe, précise que « globalement, une action *Stomp* sur un ennemi ailé ne doit pas être suivie par une action *EnemyDead* sur le même ennemi ». Les variables entre crochets permettent d'aller vérifier les endroits où le paramètre en question est égal à la valeur désignée. Cette formule s'assure aussi de prendre l'*id* de l'ennemi touché pour que l'on s'assure de regarder la prochaine action sur la même entité.

Ce jeu a permis de reproduire, en partie, les travaux de Lewis et Whitehead (2011) mais en utilisant LTL au lieu de l'engin de règles Drools.

4.3.2 THE TIMEBUILDERS : PYRAMID RISING 2

The Timebuilders : Pyramid Rising 2 est un jeu de gestion de ressources développé par BlooBuzz, une société de Québécor Média (BlooBuzz, 2012), à l'aide du moteur de jeu Unity 3D et du langage C#. Ce jeu a été développé pour PC, Mac et tablettes. Le but du jeu est de compléter plusieurs niveaux dont les objectifs varient et dont la difficulté est croissante. Pour réussir, le joueur doit ramasser des ressources, construire des bâtiments ou des unités et accomplir certains objectifs mineurs tels que payer des marchands ou amasser des trésors. Encore une fois, l'instrumentation s'est faite manuellement afin d'envoyer des événements simples et le moniteur a été directement ajouté au jeu pour une utilisation simple avec Unity. Ce jeu a été choisi grâce à une collaboration avec l'entreprise et est donc le seul n'étant pas libre de droit figurant dans nos tests. Cela a permis de tester le runtime monitoring sur un jeu développé pour le marché et détecter des problèmes que les développeurs ont rencontrés lors

```
<event>
  <name>WorkerStunAnimation</name>
  <objectID>Worker_17</objectID>
  <eventTime>4918</eventTime>
  <charType>Worker</charType>
</event>
```

Figure 4.7: Exemple d'un message XML produit par *Pyramid Rising 2*.

du développement. Toutefois, ces bogues étant déjà réglés, il a fallu manuellement réinsérer des erreurs, de la même façon que pour *Infinite Mario Bros*. Voici l'exemple d'une événement produit :

Cet événement indique que l'événement est nommé « WorkerStunAnimation » et qu'il affecte l'entité « Worker_ 17 ». Le type du personnage est aussi mentionné si nécessaire ainsi que le temps, en millisecondes, qui s'est écoulé depuis le début du jeu.

Cinq propriétés ont été développées, mais nous n'en verrons que deux.

Propriété de monitoring 4

$$\boldsymbol{G}((\forall x_1 \in /event/name : (x_1 = ElephantUpgrade)) \rightarrow \boldsymbol{X}(\boldsymbol{G}(\forall x_2 \in /event/name : \neg(x_2 = ElephantUpgrade))))$$

Cette formule précise que « globalement, lorsque l'on reçoit un événement *ElephantUpgrade*, on ne peut plus recevoir, globalement, un autre événement ElephantUpgrade ». En fait, dans le jeu, l'augmentation du moyen de transport pour obtenir des éléphants ne peut être fait qu'une fois dans un niveau. Cela implique que lorsqu'un deuxième se produit, cela va à l'encontre de l'intention du développeur.

Figure 4.8: Capture d'écran du jeu *The Timebuilders : Pyramid Rising 2*.

Propriété de monitoring 5

$$\boldsymbol{G}(\forall c \in /event[name = ActionCanceled]/objectID : (\boldsymbol{X}(\forall x_1 \in /event[objectID]/name :$$
$$\neg(x_1 = CharacterStateWaitingOrder)\boldsymbol{U}(\forall x_2 \in /event[objectID = \$c]/name :$$
$$(x_2 = CharacterStateWaitingOrder)))))$$

Cette formule indique que dans le jeu, si l'utilisateur annule une action, l'unité ne peut pas entrer dans un autre état que *CharacterStateWaitingOrder* tant qu'il n'est pas entré au moins une fois dans cet état. Le bogue observé consistait surtout en un constructeur qui tentait de continuer sa tâche alors qu'il devait tout arrêter.

Ces propriétés, ainsi que d'autres, ont permis de détecter des erreurs se produisant dans un jeu du marché fonctionnant à l'aide d'un moteur de jeu. Les bogues à surveiller étaient de vrais problèmes rencontrés lors du développement bien qu'il ait fallu les ajouter manuellement puisque ces erreurs avaient été réglées dans une version précédente du jeu.

4.3.3 PINGUS

Pingus est un clone de *Lemmings*, un jeu de puzzle libre de droit pour PC, Mac et Linux (Ruhnke, 2010) écrit en C++. L'objectif est de terminer plusieurs niveaux en guidant les pingus, des pingouins se promenant toujours droit devant, vers la sortie. Pour réussir, le joueur clique sur les pingus pour les faire changer d'état. Il est possible de leur donner un parachute, faisant ainsi en sorte que le pingouin désigné ne meurt pas en frappant le sol ou encore de les faire creuser à travers les obstacles. L'instrumentation utilise la boucle de jeu et les messages représentent l'état de tous les pingus à chaque moment. Le système et le moniteur ont aussi été séparés, le jeu produisant des messages dans un *pipe* que BeepBeep surveille afin d'obtenir les données. Les bogues dans le jeu ont été ajoutés manuellement pour obtenir certaines propriétés à observer. Voici le *template cpptempl* (Ginstrom, 2010) représentant les messages envoyés :

Cette structure permet de représenter les éléments importants de chaque pingu : identifiant, l'action entreprise, s'il est en vie ou non, sa position en x et en y, sa vélocité en x et en y et le type de sol qui se trouve sous ses pieds. De plus, certains éléments n'apparaissent qu'une fois dans le message. La valeur *deadlyvelocity* représente la vitesse de chute qui sera fatale pour le pingu et servira pour une des formules. D'autres éléments (*timestamp*, *messagenumber* et *overhead*) sont présents à des fins statistiques. Il est à noter que les seuls changements dans le code se retrouvent dans le fichier pingu_holder.cpp où se trouvent les pingus qui sont mis à jour à chaque tour de boucle.

Avec l'information obtenue sur l'état des Pingus à tout moment, il est possible d'écrire des formules surveillant l'état des différents personnages.

```
templ = <message>
  <characters>
    {% for pingu in characters.pingus %}
    <character>
      <id>{$pingu.id}</id>
      <action>{$pingu.action}</action>
      <isalive>{$pingu.isalive}</isalive>
      <position>
        <x>{$pingu.x}</x>
        <y>{$pingu.y}</y>
      </position>
      <velocity>
        <x>{$pingu.velx}</x>
        <y>{$pingu.vely}</y>
      </velocity>
      <groundtype>{$pingu.groundtype}</groundtype>
    </character>
    {% endfor %}
  </characters>
  <deadlyvelocity>{$deadlyvelocity}</deadlyvelocity>
  <timestamp>{$timestamp}</timestamp>
  <messagenumber>{$messagenumber}</messagenumber>
  <overhead>{$overhead}</overhead>
</message>\n;
```

Figure 4.9: Exemple d'un message XML produit par *Pingus*.

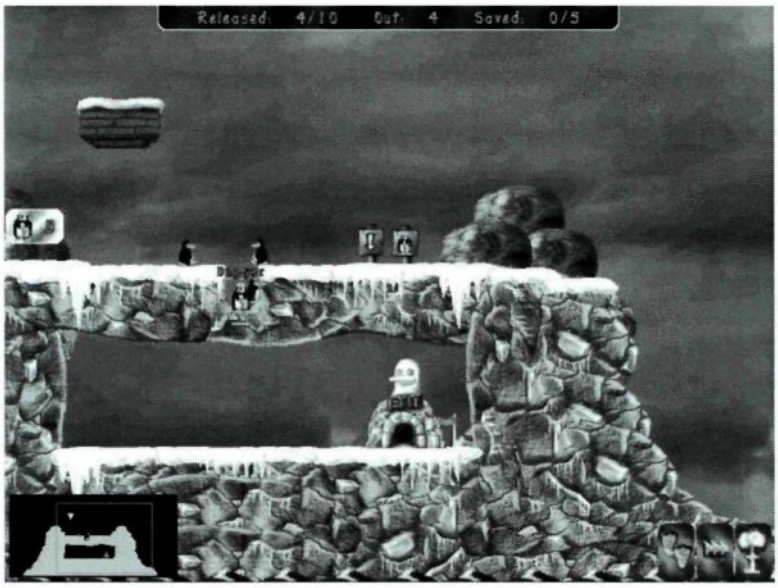


Figure 4.10: Capture d'écran du jeu *Pingus*.

Propriété de monitoring 6

$$\forall dv \in message/deadlyvelocity:$$

$$(\boldsymbol{G}(\forall c \in message/characters/character[action = faller]/id:$$

$$((\forall vy \in message/characters/character[id = \$c]/velocity/y:(vy > dv)) \rightarrow$$

$$(\boldsymbol{X}(\forall s \in message/characters/character[id = \$c]/action:$$

$$(((s = faller) \vee (s = drown)) \vee (s = splashed)))))))$$

Cette formule commence tout d'abord par sauvergarder la valeur *deadlyvelocity* à l'intérieur de la variable *dv*. Ensuite, on précise que pour chaque pingu dont l'action est de tomber et qui descend à une vitesse plus grande que la vitesse mortelle, son prochain état ne peut être que de tomber encore (*faller*), de se noyer s'il tombe dans l'eau (*drown*) ou de s'écraser (*splashed*). Cette formule est intéressante par son utilisation du quantificateur $\forall$ qui permet de vérifier tous les Pingus, ce que LTL ne permettrait pas sans l'extension LTL-FO+.

Propriété de monitoring 7

$$\boldsymbol{G}(\forall c \in message/characters/character[action = faller]/id :$$
$$(\boldsymbol{X}(\forall x \in message/characters/character[id =]/action : (\neg(x = bomber)))))$$

Dans le jeu, le pingu peut exploser, mais cela ne doit pas arriver lorsqu'il tombe. Cette contrainte dit que « globalement, chaque pingu dont l'action entreprise est de tomber ne doit pas exploser lors du prochain message ».

Ce jeu a permis de vérifier l'efficacité de la nouvelle technique d'instrumentation mise en place. L'instrumentation s'est fait beaucoup plus rapidement en ne modifiant qu'un endroit dans le code au lieu d'un nombre estimé à plus de 100. La séparation du moniteur et du système permet aussi au jeu de fonctionner au *frame rate* désiré malgré que les messages produits soient beaucoup plus gros.

4.3.4 BOS WARS

Bos Wars est un jeu de stratégie en temps réel libre de droits pour PC, Linux, Mac et BSD (Jensen et Beerte, 2010a) écrit en C++. Le jeu se déroule dans un monde futuriste et il faut tout simplement vaincre son adversaire. Pour ce faire, les joueurs doivent construire des bâtiments et des unités pour éliminer ses adversaires par la force. En utilisant la boucle de jeu, l'entièreté de l'instrumentation s'est passée dans le fichier mainloop.cpp alors que le moniteur et le système étaient séparés, communiquant par l'entremise d'un *pipe*. Au lieu d'ajouter manuellement des bogues, les propriétés écrites représentent des problèmes déjà présents dans le jeu qui avaient été reportés dans le *bug tracker* du jeu (Jensen et Beerte, 2010b). Voici la structure utilisée pour les messages :

```
templ = templ = <message>
  <units>
    {% for unit in units.tabunits %}
    <unit>
      <id>{$unit.id}</id>
      <type>{$unit.type}</type>
      <isbuilding>{$unit.isbuilding}</isbuilding>
      <player>{$unit.player}</player>
      <order>
        <action>{$unit.action}</action>
        <goal>{$unit.goal}</goal>
      </order>
      <neworder>
        <action>{$unit.neworder}</action>
        <goal>{$unit.newgoal}</goal>
      </neworder>
    </unit>
    {% endfor %}
  </units>
  <overhead>{$overhead}</overhead>
</message>\n";
```

Figure 4.11: Exemple d'un message XML produit par *Bos Wars*.

Figure 4.12: Capture d'écran du jeu *Bos Wars*.

La structure permet d'obtenir beaucoup d'informations sur tous les types d'entité du jeu, que ce soit un bâtiment, un arbre, un combattant, un rocher ou un véhicule. Il est possible d'y trouver l'identifiant de l'unité, le type d'unité, si c'est un bâtiment (puisque les bâtiments sont traités différemment), à quel joueur l'unité appartient, l'action à accomplir et le but de cette action ainsi que l'action et le but qu'aura l'unité produite (pour les bâtiments). Encore une fois, la statistique d'*overhead* est mesurée.

Les formules suivantes ont servi à détecter des bogues réels rencontrés par les développeurs du jeu :

Propriété de monitoring 8

$$\boldsymbol{G}(\forall g \in message/units/unit/order[action = UnitActionRepair]/goal :$$
$$(\forall x \in message/units/unit/order[goal = \$g]/action : (\neg(x = UnitActionResource))))$$

Le bogue tel que décrit dans le *bug tracker* faisait en sorte qu'un travailleur, dont le travail était de réparer les bâtiments autour, allait réparer les bâtiments que l'on tentait de détruire pour ramasser des ressources (bogue # 29095). La formule indique que lorsqu'un travailleur répare un bâtiment, il ne peut pas y avoir un autre travailleur qui tente de détruire ce même bâtiment. Le bâtiment est représenté par le paramètre *goal*.

Propriété de monitoring 9

$$\boldsymbol{G}(\forall g \in message/units/unit/neworder[action = UnitActionResource]/goal :$$
$$(\forall x \in message/units/unit[id = \$g]/isbuilding : (\neg(x = true))))$$

Dans le jeu, lorsque l'on crée un personnage, on peut lui donner un but qui lui sera attribué à la création. Un des problèmes rencontrés est qu'un bâtisseur doit, lorsqu'on lui donne un bâtiment comme but, le réparer alors que présentement, il va le détruire pour retirer des ressources (bogue # 29765). Cette formule précise qu'on ne peut pas permettre d'attribuer un but à une unité créée étant de ramasser les ressources d'un autre bâtiment. Ce but est représenté par le parmaètre *neworder*. Puisqu'aucune autre unité qu'un bâtisseur ne peut ramasser de ressources, il n'y a pas de raisons de spécifier le type. De plus, dans le cas où une autre unité entre dans l'état *UnitActionResource*, le moniteur détecte aussi l'échec.

Dans tous les jeux vidéo étudiés, le runtime monitoring a permis de vérifier toutes les règles spécifiées et même de détecter des problèmes rencontrés dans le développement des jeux, confirmant que BeepBeep et LTL-FO+ peuvent être utilisés dans un contexte de ce genre. Le tout s'est fait sur quatre types de jeux différents : plateforme, gestion de ressources, puzzle et stratégie en temps réel. De plus, aucun des jeux n'a démontré de ralentissement lors de l'exécution. Le prochain chapitre décrit les statistiques mesurées lors de ces tests.

Tableau 4.1: Spécifications de la machine ayant servi aux expérimentations.

Processeur	Intel Core 2 Duo CPU P7350 @ 2.00GHz x 2
Mémoire vive	4Go
Carte graphique	NVIDIA Geforce GT 130M
OS	Ubuntu 12.04 LTS 64-bit

4.4 STATISTIQUES LIÉES AUX EXPÉRIMENTATIONS

Bien que les jeux ne montrent aucun ralentissement lors des expérimentations, il faut aussi regarder l'impact de la technique en général, ce qui permettra d'extrapoler les résultats à des jeux de plus grande envergure. Nous considérons donc le nombre d'événements produits, le temps de traitement du moniteur et la surcharge représentant l'impact de la technique sur l'exécution du jeu. Tous les tests ont été effectués sur un ordinateur dont les spécifications sont données dans le tableau 4.1.

4.4.1 NOMBRE D'ÉVÉNEMENTS PRODUITS

L'instrumentation n'amène pas le même nombre d'événements produits dans chaque système. En utilisant une approche par événement, la production d'événements dépend grandement des actions du joueur, même s'il est possible que l'environnement, comme les ennemis, en produisent sans entrées fournies par l'utilisateur. L'approche utilisant la boucle de jeu donne un nombre d'événements fixe ne dépendant pas de l'utilisateur puisqu'ils sont envoyés à intervalles réguliers. *Infinite Mario Bros* produit aussi peu que 2,9 événements par seconde alors que *Pingus* déclenche, en moyenne, 167 événements par seconde. Dans le cas de *Pingus*, le nombre d'événements ne correspond pas au *frame rate* du jeu puisque le jeu rafraîchit l'affichage 40 fois par seconde mais met à jour l'état du jeu le plus souvent possible. Cela

signifie qu'il faut un minimum de 40 événements par seconde afin que le jeu ne ralentisse pas. On remarque que le jeu a assez de temps de processeur pour modifier son état plus de quatre fois ce nombre par seconde, démontrant que l'instrumentation n'affecte pas significativement le bon fonctionnement du jeu. Ce jeu représente le plus grand taux de rafraîchissement observé, les autres variant entre 40 et 60.

En moyenne, chaque événement produit par l'instrumentation est mis sous la forme d'un document XML de 700 octets. Avec ces nombres, on peut estimer une bande passante consumée par la communication entre le moniteur et le système à environ 120 kilo-octets par seconde. Il serait possible de diminuer ce nombre en compressant le XML, en diminuant la taille des noms de chaque balise ou en utilisant d'autres formes de compression.

4.4.2 SURCHARGE DU JEU

Puisque qu'il faut insérer du code à l'intérieur du jeu pour pouvoir obtenir les données requises pour le runtime monitoring, il y a aussi un coût de performance qui peut s'ajouter. Dans le cas de *Infinite Mario Bros* et *The Timebuilders : Pyramid Rising 2*, cette valeur représente autant l'instrumentation que le travail du moniteur pour calculer le verdict (puisque le moniteur est intégré dans le jeu) alors que pour *Pingus* et *Bos Wars*, cette valeur représente seulement l'instrumentation. Le tableau 4.2 représente certains éléments importants du jeu avec un *overhead* représenté à l'aide d'un pourcentage de l'exécution totale du jeu.

L'overhead des jeux, à l'exception de *Infinite Mario Bros* est comparable d'un système à l'autre. Toutefois, les messages envoyés dans les deux derniers jeux sont beaucoup plus volumineux et envoyés beaucoup plus régulièrement. De plus, en augmentant le nombre de propriétés à surveiller, l'*overhead* des deux premiers jeux augmentent alors que l'exécution restera inchangée en utilisant la boucle de jeu.

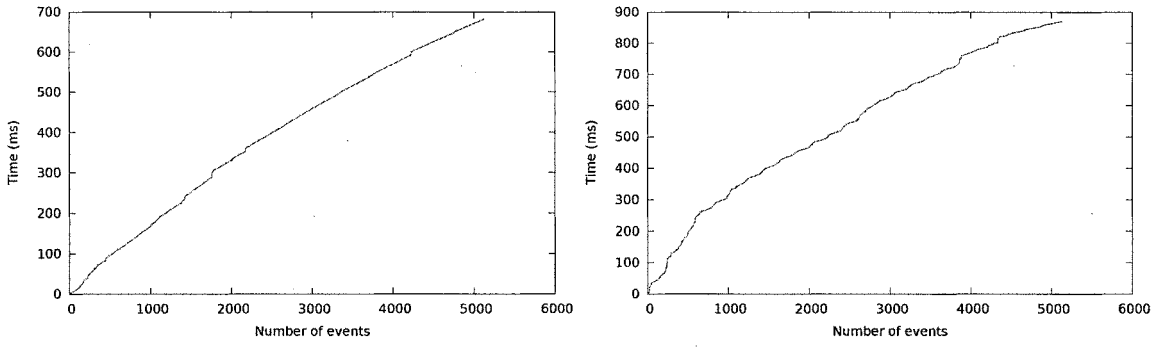
Tableau 4.2: Informations et *overhead* des jeux utilisés.

Nom du jeu	Type de jeu	Lignes de code	Frame Rate	Overhead
Infinite Mario Bros	Plateforme	6k	24fps	$\approx 1,5\%$
Pyramid Rising 2	Gestion de ressources	Inconnu	??	$\approx 5\%$
Pingus	Puzzle	40k	40fps	$\approx 6,7\%$
Bos Wars	Stratégie en temps réel	113k	30fps	$\approx 7\%$

Ces observations se concentrent sur Pingus, mais les résultats des autres jeux sont similaires. Le temps de processeur pris par l'instrumentation de Pingus augmente de façon linéaire, cumulant 2 secondes de temps d'exécution après 5000 événements, résultant à une moyenne de 0,4 milliseconde par événement. Cela démontre qu'il est possible, sur l'ordinateur utilisé pour ces expérimentations, d'instancier le *template* XML 2500 fois par seconde. Puisque le jeu boucle 167 fois par seconde, on peut déduire que l'addition du délai de 0,4 milliseconde par événement indique qu'avant l'instrumentation, le jeu se mettait à jour environ 179 fois par seconde, un *overhead* de $6,7\%$. Par contre, si on utilise les mêmes nombres sur un jeu fonctionnant à 40 fps, l'ajout de l'instrumentation modifierait ce nombre à 39,4 fps, une réduction d'environ $1,5\%$.

4.4.3 OVERHEAD DU MONITEUR

Pour bien évaluer le travail du moniteur, ce mémoire étudie si le moniteur choisi, BeepBeep (Hallé et Villemaire, 2009), est adapté pour le domaine des jeux vidéo. Le travail du moniteur consiste à ramasser les données, sous la forme d'un document XML, de regarder si la trace respecte les contraintes et de fournir un verdict. Afin de vérifier si le moniteur ne fonctionne pas à la vitesse désirée, il suffit de surveiller le tampon d'entrées. Si sa taille augmente sans cesse, cela signifie que le jeu produit des événements plus rapidement que le moniteur ne peut les traiter. Par contre, durant toutes les simulations effectuées, le tampon est toujours demeuré



(a) (b)

Figure 4.13: Temps de processeur cumulatif du moniteur pour une propriété a) Simple b) Complexe

à zéro, indiquant que le moniteur peut traiter tous les événements dès qu'ils arrivent.

La figure 4.5 représente le temps de traitement du moniteur pour deux propriétés différentes. Le premier graphique (a) représente une propriété simple de *Pingus* indiquant que tous les pingus en jeu doivent être en vie. Le temps utilisé par le processeur augmente de façon linéaire, donc ce temps est constant. En moyenne, la première propriété prend $0,13$ milliseconde de traitement, indiquant que le moniteur pourrait traiter jusqu'à 7500 événements de ce genre par seconde. La deuxième propriété (b), la plus complexe de ce mémoire, indique qu'un pingu ne doit pas survivre à une chute mortelle. Cette dernière n'est pas aussi constante, commençant par une pointe d'environ $0,3$ milliseconde par événement et un temps de traitement plus court après les 1000 premiers événements. Cette pointe peut être expliquée, puisque cette propriété exprime un fait sur les pingus en chute et les pingus tombent tous un à la suite de l'autre au début du niveau. Par contre, lorsqu'il n'y a plus de pingu en chute, le quantificateur $\forall c \in //character[action = faller]/id$ évalue le reste de la formule sur un ensemble vide, ce qui augmente la rapidité de traitement de la contrainte.

4.4.4 DÉTECTION RAPIDE DES BOGUES

Les données précédentes indiquent que le moniteur peut supporter le runtime monitoring du comportement d'un jeu vidéo et qu'il ne présente aucun problème à détecter les bogues liés à l'expérience de jeu, incluant des bogues se produisant dans le jeu Bos Wars. Dans tous les jeux, la réponse du moniteur donnait un retour d'information immédiat sur l'occurrence des bogues. L'ordinateur de référence et, supposant que chaque propriété requiert 0.3 milliseconde par événement envoyé au moniteur, une estimation conservatrice, la configuration utilisée pourraient surveiller plus de 50 propriétés aussi complexes que notre contrainte la plus longue à traiter en même temps en conservant un *frame rate* de 60 fps. Puisque la génération des événements à l'intérieur du jeu est complètement indépendante de leur traitement dans le moniteur, cette limite n'est présente que pour permettre au moniteur de ne pas être en retard sur l'état du jeu et n'a rien à voir avec le ralentissement du système. De plus, BeepBeep tel qu'implémenté, bien que fonctionnant dans un *thread* séparé du système sous surveillance, n'utilise qu'un seul *thread*. L'implémentation du *multithreading* à l'intérieur même du moniteur pourrait augmenter le nombre de propriétés pouvant être surveillées en parallèle. Chaque propriété recevrait l'événement et surveillerait sa propre contrainte, n'ayant pas à attendre que le travail de la propriété la précédant soit accompli.

CHAPITRE 5

CONCLUSION

Ce mémoire visait à démontrer la faisabilité de l'utilisation du runtime monitoring dans un contexte concernant les jeux vidéo afin de trouver une solution à la problématique rencontrée dans les tests d'assurance-qualité des jeux. Dans la grande partie des logiciels traditionnels, de nombreux bogues se retrouvent sur le marché. Les bogues sont parfois difficiles à observer et passent inaperçus lors des phases de test, le testeur peut aussi mal interpréter un comportement non désiré et ne pas le considérer comme un échec et il est pratiquement impossible de tester toutes les exécutions possibles, laissant ainsi des bogues qui ne ce sont tout simplement pas produits durant cette phase de test.

Plus précisément, ce mémoire nous a permis de découvrir qu'il était possible de surveiller efficacement un système à l'aide du runtime monitoring en établissant trois objectifs à atteindre. Le tout a commencé par l'application simple de la technique sur un jeu tout aussi simple. Il fallait trouver une façon de représenter les contraintes et LTL (Linear Temporal Logic), couramment utilisée dans les méthodes formelles, représentait une possibilité intéressante par sa présence dans le milieu de la vérification automatique. Cet objectif a été accompli en appliquant la technique sur les jeux *Infinite Mario Bros* et *The Timebuilders : Pyramid Rising 2*. Le deuxième objectif était d'implémenter une instrumentation évitant les problèmes de

l'instrumentation manuelle, longue et pénible, et de la programmation orientée aspect qui ne s'adapte pas à tous les langages. En utilisant la boucle de jeu, il devient possible d'ajouter le code d'instrumentation à un seul endroit tout en récoltant l'information des entités du jeu. Cette méthode a été appliquée dans *Pingus* avec succès. Le troisième objectif consistait à appliquer le runtime monitoring sur un jeu possédant de vrais bogues. Pour ce faire, la technique a d'abord été appliquée sur *The Timebuilders : Pyramid Rising 2*, permettant de confirmer qu'il était possible de détecter des échecs rencontrés dans un jeu prévu pour le marché. Le travail sur *Bos Wars* a permis de vérifier, avec succès, des contraintes sur des bogues encore présents dans le *bug tracker* du projet.

Pour commencer, il fallait tout d'abord s'assurer de bien connaître les limitations du runtime monitoring et l'environnement de travail apporté par les jeux vidéo. Premièrement, il a fallu établir l'état de l'art sur la détection de bogues dans le domaine des jeux vidéo malgré le nombre limité d'articles sur le sujet. Toutefois, des articles sur les différents bogues rencontrés dans les jeux vidéo (Lewis et al., 2010) et sur une implémentation déjà existante du runtime monitoring dans le domaine (Lewis et Whitehead, 2011) ont permis de mieux situer les problèmes que l'environnement des jeux apportaient pour ce genre de technique. Deuxièmement, il a fallu étudier la technique du runtime monitoring pour la mettre en œuvre. Des articles comme celui de Leucker et Schallhart (2009) démontrent que le runtime monitoring est bien adapté pour le travail de ce mémoire. Le choix du moniteur s'est arrêté sur BeepBeep (Hallé et Villemaire, 2009) puisqu'il permettait d'obtenir de l'information spécifique pour chaque événement, élément souvent absent des outils de runtime monitoring.

Ce travail de recherche a permis d'établir des points importants sur le travail à accomplir. Bien que possédant plusieurs caractéristiques idéales pour le runtime monitoring, certaines limitations se présentaient. L'instrumentation manuelle, courante dans la vérification à l'exécution, n'est pas idéale. Étant parmi les systèmes les plus complexes, l'instrumentation manuelle

devient pénible et très intrusive puisqu'il faut chercher et modifier tous les endroits requis pour envoyer les messages. L'autre méthode la plus courante, la programmation orientée aspect (Nusayr, 2008), est très dépendante du langage, ce qui n'est pas adapté puisque les jeux vidéo utilisent de nombreux langages de programmation différents. De plus, puisque le runtime monitoring consiste en l'exécution d'un moniteur en même temps que le système, il faut une méthode ne demandant pas trop de temps de processeur afin de ne pas déranger l'exécution du jeu où le ralentissement n'est pas désiré.

L'approche utilisée pour éviter ces problèmes consiste à minimiser l'intrusion à un seul fichier du projet. La boucle de jeu, où l'état du jeu change un certain nombre de fois par seconde, est l'endroit idéal puisque l'on peut récupérer l'information désirée sur toutes les entités, ce qui rend l'instrumentation beaucoup moins intrusive. Au lieu d'envoyer des événements lorsque quelque chose se produit, on envoie l'état du jeu à chaque mise à jour de celui-ci au moniteur. Il est aussi important de séparer le moniteur et le système afin de permettre une indépendance d'exécution et de langage (Lewis et Whitehead, 2011). Le nombre de propriétés pouvant être surveillées en même temps dépend ainsi du traitement du moniteur et non pas de l'impact du temps de processeur retiré au jeu. Pour permettre la communication entre le jeu et le moniteur, on ouvre un *pipe* où les messages, sous forme d'un XML, sont transmis du système vers le moniteur.

Les résultats obtenus lors des expérimentations sont prometteurs. Il a été possible de détecter des bogues rencontrés dans quatre jeux de quatre différents types : plateforme, gestion de ressources, puzzle et stratégie en temps réel. Cela offre une bonne couverture des types de jeu pour ensuite présumer que la plupart des jeux peuvent utiliser le runtime monitoring. Aucun ralentissement n'a été rencontré, indiquant que le runtime monitoring n'affectait pas, pour les jeux utilisés, l'exécution du système. Il a même été démontré qu'il serait possible de surveiller plus de 50 propriétés, chacune d'un degré de complexité égal à la plus complexe de celles

écrites pour les expérimentations de ce mémoire, en simultané.

Cette technique pourrait toutefois être améliorée. En recherchant les particularités retrouvées dans les jeux vidéo, il a été possible de remarquer l'importance du suivi des bogues à l'aide d'un *bug tracker*. Il serait intéressant d'insérer automatiquement les bogues rencontrés dans un logiciel de ce genre pour qu'ils puissent être traités le plus rapidement possible. Afin d'éviter une répétition des mêmes problèmes, une méthode pour regrouper des problèmes selon la trace et le type du bogue aperçu permettrait d'éviter les entrées semblables dans le *bug tracker*. Du côté du moniteur, puisque BeepBeep n'utilise qu'un seul *thread*, il serait possible d'augmenter le nombre de propriétés à surveiller en un coup en travaillant de façon *multithread*.

À terme, les techniques présentées dans ce mémoire pourront permettre de faciliter le dépistage des bogues dans les jeux vidéo et de diminuer le temps passé à rechercher les bogues.

BIBLIOGRAPHIE

Avizienis, A., J.-C. Laprie, B. Randell, et C. Landwehr. 2004. « Basic concepts and taxonomy of dependable and secure computing ». T. 1, p. 11–33. Dependable and Secure Computing, IEEE Transactions on.

Bartel, R. 2011. Dragon age II PC and Mac patch 1.02 is now live. `http://social.bioware.com/forum/Dragon-Age-II/Dragon-Age-II-PC-Technical-Self-Help/Dragon-Age-II-PC-and-Mac-Patch-102-is-now-live-7048601-1.html`.

Basin, D., F. Klaedtke, et S. Müller. 2010. *Policy Monitoring in First-Order Temporal Logic*. Coll. Touili, T., B. Cook, et P. Jackson, éditeurs, Coll. « *Computer Aided Verification* ». T. 6174, série *Lecture Notes in Computer Science*, p. 1–18. Springer Berlin Heidelberg.

Bauer, A., M. Leucker, et C. Schallhart. 2011. « Runtime verification for LTL and TLTL », *ACM Trans. Softw. Eng. Methodol.*, vol. 20, no. 4, p. 14 :1–14 :64.

Berglund, A., S. Boag, D. Chamberlin, M. F. Fernandez, M. Kay, J. Robie, et J. Siméon. 2010. Xml path language (xpath) 2.0 (second edition). `http://www.w3.org/TR/xpath20/`.

Bettenburg, N., S. Just, A. Schröter, C. Weiss, R. Premraj, et T. Zimmermann. 2008. « What makes a good bug report ? ». In *Proceedings of the 16th ACM SIGSOFT International*

Symposium on Foundations of Software Engineering. Coll. « SIGSOFT '08/FSE-16 », p. 308–318. ACM.

BlooBuzz. 2012. The Timebuilders : Pyramid Rising 2. `http://www.bloobuzz.com/en/games/pyramid-rising-2/`.

Board, I. S. 1987. *IEEE Standard for Software Unit Testing : An American National Standard.*

Bodden, E., L. Hendren, P. Lam, O. Lhoták, et N. Naeem. 2007. *Collaborative Runtime Verification with Tracematches*. Coll. Sokolsky, O. et S. Taşıran, éditeurs, Coll. « *Runtime Verification* ». T. 4839, série *Lecture Notes in Computer Science*, p. 22–37. Springer Berlin Heidelberg.

Bonacina, M. P. 2010. « On theorem proving for program checking : Historical perspective and recent developments ». In *Proceedings of the 12th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming*. Coll. « PPDP '10 », p. 1–12. ACM.

Clarke, E. M. J., O. Grumberg, et D. A. Peled. 1999. *Model Checking*. The MIT Press.

Colombo, C., G. J. Pace, et G. Schneider. 2009. « Larva — safer monitoring of real-time Java programs (tool paper) », *Third IEEE International Conference on Software Engineering and Formal Methods (SEFM'09)*, p. 33–37.

d'Amorim, M. et K. Havelund. 2005. « Event-based runtime verification of Java programs », *SIGSOFT Softw. Eng. Notes*, vol. 30, no. 4, p. 1–7.

Daniel, M. 2011. Star wars : The old republic beta test attracts 2 million players. `http://massively.joystiq.com/2011/12/05/star-wars-the-old-republic-beta-test-attracts-2-million-players/`.

Falcone, Y., J.-C. Fernandez, et L. Mounier. 2008. *Synthesizing Enforcement Monitors wrt. the Safety-Progress Classification of Properties*. Coll. Sekar, R. et A. Pujari, éditeurs, Coll. « *Information Systems Security* ». T. 5352, série *Lecture Notes in Computer Science*, p. 41–55. Springer Berlin Heidelberg.

Farokhmanesh, M. 2013. Skyrim update 1.9 adds legendary skills and difficulty, 'removes' level cap. `http://www.polygon.com/2013/4/8/4197552/skyrim-update-1-9-adds-legendary-skills-and-difficulty-removes-level`.

Foltz, B. 2011. The game loop and frame rate management. `https://www.brandonfoltz.com/downloads/tutorials`.

Foster, M. 2013. Blizzard to launch Diablo 3's reaper of souls closed beta before the end of the year. `http://massively.joystiq.com/2013/11/20/blizzard-to-launch-reaper-of-souls-closed-beta-before-the-end-of/`.

Gaudiosi, J. 2012. New reports forecast global video game industry will reach $82 billion by 2017. `http://www.forbes.com/sites/johngaudiosi/2012/07/18/new-reports-forecasts-global-video-game-industry-will-reach-82-billion-by-2017/`.

Ginstrom, R. 2010. cpptempl : A string templating library. `http://ginstrom.com/scribbles/2010/10/30/cpptempl-a-template-language-for-c/`.

Hallé, S. et R. Villemaire. 2009. *Browser-Based Enforcement of Interface Contracts in Web Applications with BeepBeep*. Coll. Bouajjani, A. et O. Maler, éditeurs, Coll. « *Computer Aided Verification* ». T. 5643, série *Lecture Notes in Computer Science*, p. 648–653. Springer Berlin Heidelberg.

Huth, M. et M. Ryan. 2004. *Logic in Computer Science : Modelling and Reasoning About Systems*. New York, NY, USA : Cambridge University Press.

Jensen, T. P. et F. Beerte. 2010a. Bos wars. `http://www.boswars.org/index.shtml`.

———. 2010b. Bos wars - anomalies. `http://savannah.nongnu.org/bugs/?group=stratagus-bos`.

Kiczales, G., E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, et W. Griswold. 2001. *An Overview of AspectJ*. Coll. Knudsen, J., éditeur, Coll. « *ECOOP 2001 - Object-Oriented Programming* ». T. 2072, série *Lecture Notes in Computer Science*, p. 327–354. Springer Berlin Heidelberg.

Klement, K. C. 2005. Propositional logic. `http://www.iep.utm.edu/prop-log/`.

Leucker, M. et C. Schallhart. 2009. « A brief account of runtime verification », *The Journal of Logic and Algebraic Programming*, vol. 78, no. 5, p. 293–303.

Levy, L. et J. Novak. 2010. *Game QA and Testing*. Coll. « Game Development Essentials ». Delmar, Cengage Learning, First édition.

Lewis, C. et J. Whitehead. 2010. « Runtime repair of software faults using event-driven monitoring. ». In Kramer, J., J. Bishop, P. T. Devanbu, et S. Uchitel, éditeurs, *ICSE (2)*, p. 275–280. ACM.

Lewis, C. et J. Whitehead. 2011. « Repairing games at runtime or, how we learned to stop worrying and love emergence », *Software, IEEE*, p. 53–59.

Lewis, C., J. Whitehead, et N. Wardrip-Fruin. 2010. « What went wrong : A taxonomy of video game bugs ». In *Proceedings of the Fifth International Conference on the Foundations of Digital Games*. Coll. « FDG '10 », p. 108–115. ACM.

Meredith, P. O., D. Jin, D. Griffith, F. Chen, et G. Roşu. 2011. « An overview of the MOP runtime verification framework », *International Journal on Software Techniques for Technology Transfer*, p. 249–289.

Minker, J. 2000. *Introduction to Logic-Based Artificial Intelligence*. Coll. Minker, J., éditeur, Coll. « *Logic-Based Artificial Intelligence* ». T. 597, série *The Springer International Series in Engineering and Computer Science*, p. 3–33. Springer US.

Mogul, J. C. 2006. « Emergent (mis)behavior vs. complex software systems ». In *IN EUROSYS*, p. 293–304. ACM.

Moura, D., M. S. el Nasr, et C. D. Shaw. 2011. « Visualizing and understanding players' behavior in video games : Discovering patterns and supporting aggregation and comparison ». In *ACM SIGGRAPH 2011 Game Papers*. Coll. « SIGGRAPH '11 », p. 2 :1–2 :6. ACM.

NIST. 2002. The economic impacts of inadequate infrastructure for software testing. Rapport, National Institute of Standards and Technology.

Nusayr, A. 2008. « AOP as a formal framework for runtime monitoring ». In *Proceedings of the 2008 Foundations of Software Engineering Doctoral Symposium*. Coll. « FSEDS '08 », p. 25–28.

Nusayr, A. et J. Cook. 2009. « Using AOP for detailed runtime monitoring instrumentation ». In *Proceedings of the Seventh International Workshop on Dynamic Analysis*. Coll. « WODA '09 », p. 8–14, New York, NY, USA. ACM.

Persson, M. 2006. Infinite Mario Bros ! `https://mojang.com/notch/mario/`.

Purchese, R. 2012. Games are arguably the most sophisticated and complex forms of software out there these days. `http://www.eurogamer.net/articles/2012-09-28-games-are-arguably-the-most-sophisticated-and-complex-forms-of-software-out-there-these-days`.

Ruhnke, I. 2010. Pingus. `http://pingus.seul.org/welcome.html`.

Shirai, A., E. Geslin, et S. Richir. 2007. « Wiimedia : Motion analysis methods and applications using a consumer video game controller ». In *Proceedings of the 2007 ACM SIGGRAPH Symposium on Video Games*. Coll. « Sandbox '07 », p. 133–140. ACM.

Sloper, T. 2013. How to write a game bug report. `http://sloperama.com/advice/faq75.htm`.

Starr, K. 2007. Testing video games can't possibly be harder than an afternoon with xbox, right ? `http://www.seattleweekly.com/2007-07-11/news/testing-video-games-can-t-possibly-be-harder-than-an-afternoon-with-xbox-right/`.

Steinicke, F., G. Bruder, K. Hinrichs, et A. Steed. 2009. « Presence-enhancing real walking user interface for first-person video games ». In *Proceedings of the 2009 ACM SIGGRAPH Symposium on Video Games*. Coll. « Sandbox '09 », p. 111–118. ACM.

Stolz, V. et E. Bodden. 2006. « Temporal assertions using Aspectj », *Electron. Notes Theor. Comput. Sci.*, vol. 144, no. 4, p. 109–124.

TechExcel. 2010. Devtrack. `http://techexcel.com/products/devtrack/`.

Varvaressos, S., K. Lavoie, A. Blondin Massé, S. Gaboury, et S. Hallé. 2014. « Automated bug finding in video games : A case study for runtime monitoring ». In *Software Testing, Verification and Validation (ICST), 2014 IEEE Seventh International Conference on*.

Varvaressos, S., D. Vaillancourt, S. Gaboury, A. Blondin Massé, et S. Hallé. 2013. « Runtime monitoring of temporal logic properties in a platform game ». In Legay, A. et S. Bensalem, éditeurs, *Runtime Verification*. T. 8174, série *Lecture Notes in Computer Science*, p. 346–351. Springer Berlin Heidelberg.

Vijayaraghavan, G. et C. Kaner. 2003. « Bug taxonomies : Use them to generate better tests ». Coll. « Star East 2003 ». p. 1–40. NSF.

Watson, A. B. 2013. How to write up good video games concerns (report a bug). http://www.andrewbrettwatson.com/index.php/interesting/204-how-to-write-up-good-video-games-concern-report-a-bug.

Watts, S. 2012. The worst video game glitches. http://www.1up.com/features/worst-video-game-glitches.