Behaviour-based Cheat Detection in Multiplayer Games with Event-B

HaiYun Tian*, Phillip J. Brooke, Anne-Gwenn Bosser

School of Computing, Teesside University, Middlesbrough, UK, TS1 3BA
H.Tian@tees.ac.uk, pjb@scm.tees.ac.uk, A.G.Bosser@tees.ac.uk

Abstract. Cheating is a key issue in multiplayer games as it causes unfairness which reduces legitimate users' satisfaction and is thus detrimental to game revenue. Many commercial solutions prevent cheats by reacting to specific implementations of cheats. As a result, they respond more slowly to fast-changing cheat techniques. This work proposes a framework using Event-B to describe and detect cheats from server-visible game behaviours. We argue that this cheat detection is more resistant to changing cheat techniques.

Keywords: Cheat detection, multiplayer games, Event-B

1 Introduction

Multiplayer games give players a sense of reality and engagement more so than in single-player games. They have gained considerable popularity in the entertainment world [5, 10, 19]. Cheating is a major concern for many game developers as it reduces the fairness of games, damages the expected game experience and thus decreases revenue [7, 10, 17]. "Cheating" refers to any game behaviour that players use to achieve an unfair advantage and/or a target that they are not supposed to [15]. Cheating may exceed the possible bounds of human capability, e.g., Aimbot, Spinbot, or provide "extra-sensory perception" such as seeing through opaque objects (commonly called "wallhacking") as well as learning about the hidden information (ESP) [23]. Moreover, cheating has grown to such an extent that not only are private hackers involved but also some companies commercially thrive by offering cheat techniques [10, 11].

Game developers face an up-hill battle with cheat developers [17]. Many commercial solutions (e.g., DMW [13], GameGuard [14], VAC [21], etc.) act reactively by discovering and studying unknown cheat techniques then developing countermeasures, as illustrated in Figure 1. However, games can remain vulnerable to particular cheats in this defense process. This work is inspired by Laurens et al. [17], a proof-of-concept solution that calculates game behaviours for indications of cheating. This work attempts to formalise the description of game behaviours using Event-B and provides a behaviour analysis method for detecting cheating. Although we specify some behaviours within a particular game,

* Corresponding author.

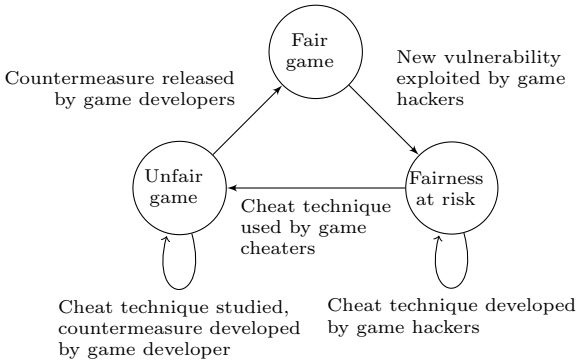

Fig. 1. Traditional cheat defence loop

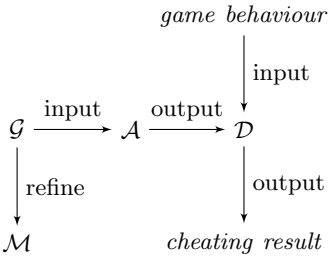

Fig. 2. $\mathcal{A}$'s environment

we are not attempting to specify a complete game. As a result, matters such as deadlock, fairness and liveness not within the scope of this work.

The rest of this paper is structured as follows. Section 2 presents our approach to modelling cheating behaviour in Event-B; section 3 introduces the actual production of cheat detectors. Section 4 validates the framework using an example game. Section 5 discusses related work before we conclude in section 6.

2 Behaviour-based Cheat Detection with Formal Methods

The framework presented in this paper contains both a method for modelling game behaviour and a procedure for producing behaviour-based cheat detectors. This framework, illustrated in Figure 2, can be divided into two sub-systems, S_1 and S_2:

$$S_1 = (\mathcal{M}, \mathcal{G}, \mathcal{A}, \mathcal{D}) \qquad S_2 = (\mathit{game\ behaviour}, \mathcal{D}, \mathit{cheating\ results})$$

S_1 produces cheat detectors: algorithm $\mathcal{A}$ uses $\mathcal{G}$, a derivation of a base model $\mathcal{M}$, to produce a cheat detector $\mathcal{D}$. S_2 uses a detector produced by S_1 to measure the cheating behaviours of players. We next consider how we might model cheating behaviours before introducing Event-B in section 2.2.

2.1 Cheating Behaviour Modelling

For now, we consider only games based on a client/server architecture. That is, we have multiple players who send commands to and receive feedback and updates from the server.

A player's behaviour is usually a thoughtful response to the current game state. This game state is distributed by the game server to all involved players, and their responses result in the subsequent game state. We write a game state as

$$game\_state = \left\{ \begin{array}{c} player_1.local\_state, \\ \vdots, \\ player_n.local\_state \end{array} \right\}$$

i.e., a *game_state* is a collection of client states reported by all involved clients $player_{1..n}$ to their game server at an instant. Thus, a particular game behaviour can be described as

$$game\_behaviour = (game\_state, game\_state')$$

This means that a game behaviour is an ordered pair of antecedent and consequent game states. A *player*'s behaviour is also an ordered pair

$$player\_behaviour = (game\_state, game\_state'(player)).$$

player represents a player and $game\_state'(player)$ stands for the *player*'s response to the antecedent *game_state*.

We use predicates to define these game behaviours. Let *STATE* be a set containing all possible client states and *PLAYER* a set of all involved players. We give a predicate, $P1$, for describing general player behaviour below.

$$P1((game\_state, game\_state'(player))) = \\ game\_state, game\_state' \in PLAYER \rightarrow STATE \ \wedge \ player \in PLAYER$$

P1 is very abstract. Suppose the game involves three possible player responses, moving, aiming and firing. We can then refine *STATE* to $\{position, aim, fire\}$. At this level of abstraction, we can also specify a particular cheat, a "triggerbot" which automatically fires whenever an opponent is located such that the opponent is likely to be hit. This gives its users an advantage over legitimate opponents of being the first to fire. We can describe triggerbotting behaviour in the predicate $P2$ as defined below.

$$\begin{aligned} P2((game\_state, game\_state'(player))) = P1 \ \wedge \\ (\exists \ peer \in PLAYER \ \wedge \ peer \neq player \\ \wedge \ game\_state'(player).aim = game\_state(peer).location) \\ \Rightarrow game\_state'(player).fire = true \end{aligned}$$

The predicate $P2$ includes $P1$ in its conjunction: this means that all behaviour described by $P2$ are permitted by $P1$. In addition, $P2$ describes that, whenever an opponent is at the point of aim, the player immediately fires. It is very likely that fair (i.e., non-cheating) players can achieve immediate fire responses too, but we suggest that they do it less frequently than a cheater since a triggerbot allows its users to exceed the typical bounds of human capability. Most widespread game cheats do not employ an 'alien' behaviour, a behaviour out of the bounds of $\{P1\}$. Otherwise, cheat detection would not be difficult since the message from the client would be obviously unacceptable. As a result, many cheats cause more effective play than a fair player during a game period. However, a cheat that exhibits behaviours identical (in probabilistic terms) to a fair player cannot be detected by our system.

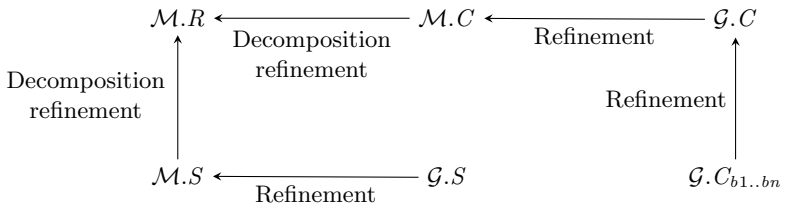

Fig. 3. Derivation from base model

Using formal languages to describe a game's behaviour can ensure the consistency in different developers' individual understanding of cheating behaviour at different abstraction levels. For example, the members of a development team (e.g., game designers, game developers, security specialists) can use a formal description to ensure a consistent description of particular cheats, as well as the overall design (in common with existing formal methods approaches).

2.2 Cheating Behaviour in Event-B

Event-B [3] is chosen in this particular work for two reasons. Firstly, many commercial games use a vast amount of data associated with world simulation, and thus modelling even an abstraction of these simulations would encounter complicated data structures. Event-B is a state-rich formalism which suits this requirement. Other options included Z [22], B [1] and Circus [9]. Secondly, Event-B has a toolkit Rodin [2, 4]. Rodin contains a model editor, an obligation generator, an obligation viewer, an obligation automatic proof, comprehensive model analysis, etc., together with a friendly user interface.

Event-B models can be constructed in an arbitrary way. As our contribution involves $\mathcal{A}$, a mechanical procedure for producing cheat detectors, we need to constrain these models so that $\mathcal{A}$ is able to work. This work uses constraints on both refinement architecture and machine refinement to ensure $\mathcal{G}$'s compatibility to $\mathcal{A}$ via an abstraction $\mathcal{M}$, the base model.

We refine the machines as shown in Figure 3. The base model $\mathcal{M}$ acts as the top abstraction of client/server game systems, providing the game model with a structure suitable for subsequent processing by $\mathcal{A}$. Decomposition refinement is a technique of describing parallelism in Event-B due to Butler [8]: $\mathcal{M}$ uses it to describe synchronisation between game servers and game clients. Details of a specific game and its cheats can be added by refinements from $\mathcal{M}$ to $\mathcal{G}$ via refinement. The machine $\mathcal{G}.S$, the model of game servers, is refined from $\mathcal{M}.S$. The machine $\mathcal{G}.C$ refines $\mathcal{M}.C$, modelling the game clients of a specific game. The machines $\mathcal{G}.C_{b1..bn}$ refine $\mathcal{G}.C$, modelling different types of cheating game clients.

During this refinement, we instantiate some variables listed in Table 1. For conciseness, this report omits some details; they can be found in [20] (along with full versions of the machines presented shortly). Machine $\mathcal{G}.C$ can be defined as

Var	Machine component description
$\mathcal{S}_f$	A list of machine state variables.
$\mathcal{I}_f(V)$	A list of invariants on V, implicitly conjoined.
$\mathcal{E}_f$	A label for a machine event.
$\mathcal{X}_f$	A list of event parameters.
$\mathcal{P}_f(V)$	A list of guards on V, implicitly conjoined.
$\mathcal{Q}_f(V)$	A list of actions on V.

Table 1. Machine component variables

Variable	Machine component
$\mathcal{E}_f$ (event name)	*play*
$\mathcal{P}_f$ (event para)	*pos*
$\mathcal{P}_f$ (event guard)	$position(cstate) = pos$

Table 2. Example of component variable assignments (client)

follows:

Machine $\mathcal{G}.C$
Refines $\mathcal{M}.C$
Sees $\mathcal{G}.Cxt$
Variables $buffer, clients, game\_situation, local\_state, own\_ID, \mathcal{S}_f$
Invariants $\ldots, \mathcal{I}_f(\mathcal{S}_f, buffer, clients, game\_situation, local\_state, own\_ID)$

Events
$initialisation \mathrel{\widehat{=}} \mathcal{Q}_f(\mathcal{S}_f) \ldots$ **end**
$\ldots$
$\mathcal{E}_f \mathrel{\widehat{=}}$
refine *client_act*
any $cstate, \mathcal{X}_f$ **where**
$clientID(cstate) = own\_ID$
$cstate \in CLIENT\_STATE$
$\mathcal{P}_f(\mathcal{S}_f, \mathcal{X}_f, buffer, clients, game\_situation, local\_state, own\_ID)$
then
$local\_state := cstate$
$\mathcal{Q}_f(\mathcal{S}_f, \mathcal{X}_f, buffer, clients, game\_situation, own\_ID)$
End
End

The full text of this machine (and others given in this work) has been checked in Rodin; the version presented here is a shortened version from Rodin's LaTeX plugin.

Suppose we have a game where avatars can change their positions. Assume the model context $\mathcal{G}.Cxt$ has a fresh carrier set MAP containing all possible positions that avatars can move to in the game; a set $CLIENT\_STATE$ contains all possible client state; and one axiom $position \in CLIENT\_STATE \to MAP$.

Variable	Machine component
$\mathcal{E}_f$ (event name)	*wallhack*
$\mathcal{P}_f$ (event guard)	$\exists\, opponent\_state \in game\_situation$ $\wedge\ clientID(opponent\_state) \neq own\_ID$ $\wedge\ position(opponent\_state) \notin VISIBLE(local\_state)$ $\wedge\ (DISTANCE(pos, position(opponent\_state))$ $\quad < DISTANCE(position(local\_state), position(opponent\_state)))$

Table 3. Component variable assignments for wallhacking example

An instance of $\mathcal{E}_f$ can be made using Table 2 so that

play $\widehat{=}$
refine *client_act*
any *cstate*, *pos* **where**
$clientID(cstate) = own\_ID$
$cstate \in CLIENT\_STATE$
$position(cstate) = pos$
then
$local\_state := cstate$
End

Now consider the cheat "wallhacking". This cheat allows players to see through opaque objects. A behavioural characteristic of this cheat can be described that

Antecedent state: An opponent is not visible to a wallhacker.
Player's response: The wallhacker approach that opponent.

This characteristic can be described in Table 3 by adding a fresh guard to *play* in a refinement. This guard involves some fresh constants and variables. Briefly, *game_situation* is defined in the base model $\mathcal{M}$; it is a set containing the up-to-date game state. These constants are defined in the Event-B context. For example, the constant *VISIBLE* represents the game function that calculates the up-to-date visible zone for a game client, and *DISTANCE* calculates the distance between two map locations.

The event *wallhack* refines *play* by adding a fresh guard, which describes that a player can approach to a legally invisible opponent. A wallhacking client machine can then be composed as the following.

Machine $\mathcal{G}_{TankWar}.C_{Wallhack}$
Refines $\mathcal{G}_{TankWar}.C$
Sees $\mathcal{G}_{TankWar}.Cxt$
Variables ...
Invariant ...
Event ...
wallhack $\widehat{=}$ ... **End**
End

As more behavioural characteristics are identified, more events like *wallhack* can

be added, and better cheat detection would is achieved using the algorithm $\mathcal{A}$ (described in the next section).

3 Production of Cheat Detector

This work aims at mechanically producing behaviour-based cheat detectors. We now describe our approach to detection, given the Event-B models outlined above.

Suppose we embed a specific model of cheating behaviour into a robot player, *robot* (e.g., the wallhacker behaviour or the triggerbot behaviour). Assume we now ask the detector to examine a game player, *player*. The detector first records a sequence of *player*'s behaviours, which are not necessarily adjacent in time or captured at the same interval, and then runs *robot* using the antecedent game state of each behaviour and collects *robot*'s response to the same sequence. Recall that a player's game behaviour is an ordered pair of the antecedent game state and the player's response (consequent client state). It is very likely that *robot* has a nondeterministic choice in its response. Thus, the detector collects from the *robot* a set that contains all its possible responses in that context. This is repeated until all collected *player* behaviours are used. In the end, the detector has a sequence of responses from *player* and a corresponding sequence of response sets from *robot*. The proportion of *player*'s response contained in *robot*'s corresponding response set is the rate that *player* behaves like this cheat.

To calculate whether or not *player* can make the same response as *robot*, we introduce the function *Exam* below.

$$Exam((game\_state, game\_state'), player, robot) = \begin{cases} 1 & \text{if } game\_state'(player) \in \text{robot's response set to } game\_state \\ 0 & \text{otherwise.} \end{cases}$$

The detector $\mathcal{D}$ uses *Exam* to assess monitored game behaviour. Recall that a game behaviour is a game state transition and described as $(game\_state, game\_state')$. Let tr be a sequence of game behaviour and k be the length of tr:

$$tr = \langle (tr_1.game\_state, tr_1.game\_state'), \ldots, (tr_k.game\_state, tr_k.game\_state') \rangle$$

That is, tr is a sequence of observations of game behaviours, or a sequence of pairs of states. The expressions $tr_i.game\_state$ and $tr_i.game\_state'$ represent the first element (antecedent game state) and the second element (consequent game state) of a particular transition tr_i.

Rather than calculating a single numeric matching rate between *player* and *robot*, our detector calculates how much *player* acts as *robot* for each leading subsequence of tr and returns a sequence of matching rates. This allows consideration of trends through a particular game, as it carries more information than a single final rate. For example, when a player finishes at 15%, but stays above 60% more than half time of the game, this player might be suspicious.

Let $rate_i$ be the matching rate for the subsequence $\langle tr_1, \ldots, t_k \rangle$. We can calculate $rate_i = \dfrac{\sum_{m=1}^{i} Exam(tr_m, player, robot)}{i}$ $[i \in 1..k]$. This describes that $rate_i$ is the proportion of the transitions in which *player* matches *robot* to the total transitions that are collected until t_i. We can use $rate_i$ to define a function *rateDst* to calculate a rate sequence

$$rateDst(\langle tr_1, \ldots, tr_k \rangle, player, robot) \mathrel{\widehat{=}} rate_1, \ldots, rate_k$$

Thus,

$$rateDst(\langle tr_1, \ldots, tr_k \rangle, player, robot) = \langle \frac{\sum_{m=1}^{1} Exam(tr_1, player, robot)}{1}, \ldots, \frac{\sum_{m=1}^{k} Exam(tr_m, player, robot)}{k} \rangle$$

For example, a sequence of game behaviour (game state transition) is captured as $tr' = \langle tr_1, tr_2, tr_3, tr_4 \rangle$. Given the following

$$Exam(tr_1, player, robot) = 1,\ Exam(tr_3, player, robot) = 0,$$
$$Exam(tr_2, player, robot) = 0,\ Exam(tr_4, player, robot) = 1$$

then $rateDst(\langle tr_1, tr_2, tr_3, tr_4 \rangle, player, robot) = \langle 100\%, 50\%, 33\%, 50\% \rangle$.

So the function *rateDst* takes the input of game behaviour sequence and produces matching rates in the same sequence. The algorithm $\mathcal{A}$ uses *rateDst* to construct the detector $\mathcal{D}$, and is defined below.

Algorithm: $\mathcal{A}$
input : $\mathcal{G}$, a game model for *game*.
output: $\mathcal{D}$, a cheat detector for *game*.
begin
  $\mathcal{D} \mathrel{\widehat{=}}$ **begin**
    input : *player*, a game client.
    input : tr, a sequence of server-side game state transitions.
    output: $ratedsr_{c1..cn}$, matching distributions for $\mathcal{G}.C_{b1}..\mathcal{G}.C_{bn}$.
    $ratedsr_{c1} = rateDst(tr, player, \mathcal{G}.C_{b1})$
    ⋮ ⋮
    $ratedsr_{cn} = rateDst(tr, player, \mathcal{G}.C_{bn})$
  end
end

Using the example above, $\mathcal{A}$ replaces *robot* with the machine $\mathcal{G}.C_{b1..bn}$, which describes the behaviours of different cheats. Thus, a produced detector $\mathcal{D}$ is able to detect these cheats by comparing the possible behaviours specified in the formal model. As presented in Figure 2, the algorithm $\mathcal{A}$ takes a behaviour

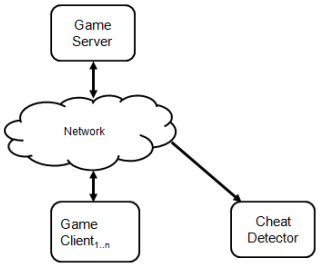

Fig. 4. Deployment of detector (1)

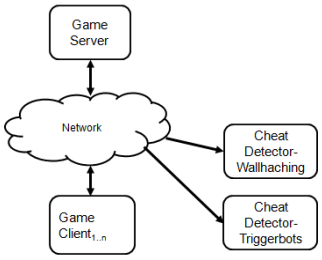

Fig. 5. Deployment of detector (2)

model $\mathcal{G}$ and returns a detector $\mathcal{D}$. The resulting detector can monitor players for the cheats that $\mathcal{G}$ specifies.

Importantly, it is necessary to discuss the criteria of judging a player using the output of our cheat detectors. Judging a player does not rely on the player's matching rates alone. We must consider the individual rates in the context of all the players' rates.

For example, when most players have rates between 5% and 15%, some players may always stay at about 30%: these latter players are suspects. We must also consider the cases that our detectors are wrongly specified, i.e., the predicates in the Event-B model are ineffective. This could result in no cheats being detected (false negatives) or too many fair players being identified as cheats (false positives). Another interesting case concerns most players engaging in cheating: this results in many high rates and it becomes difficult for an automated process to suitably identify them. The detection performance is determined by the quality of cheating behaviour knowledge that the framework users accumulate before embedding them in the framework as predicates.

3.1 Merits of Implementation

Our cheat detection can handle increasing amounts of work in a scalable manner. $\mathcal{A}$'s detectors only need the data packets that the game server distributes to game clients for maintaining game consistency, and requests neither extra information nor any computing service from the game server and clients. The number of detectors working for a game simultaneously has no impact on the performance of games, provided that the server broadcasts the relevant packets onto their network. When dealing with a number of cheats, rather than using a single 'giant' detector for all them as Figure 4 shows, we could produce one detector per cheat. Assume there are two cheats, wallhacking and triggerbots: a possible deployment can be as shown in Figure 5 with, one node for one cheat. When a new cheat detector is produced, it can be plugged into the network as a new independent node.

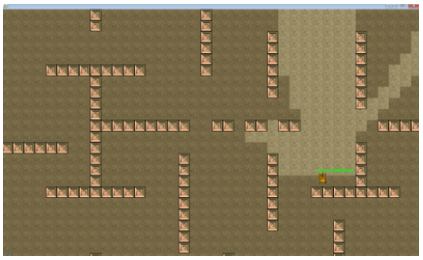

Fig. 6. Visibility zones

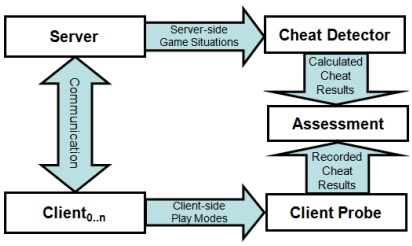

Fig. 7. Validation design

4 Validation of the framework

This framework is validated by an experiment. An example game, *TankWar* (more fully described in [20]), was used to test the resulting detector with human volunteers. It is simple by contrast with many commercial games. But it contains most important elements of first-person shooter games, e.g., surviving, attacking, limited vision, etc. More importantly, $\mathcal{M}$ is a generic client/server architecture game model, and it is designed for deriving behaviour model $\mathcal{G}$ for the games of this tier, even though *TankWar* is relatively small. Therefore, we suggest that it is a suitable proof-of-concept test of its applicability for larger games.

TankWar, is a real-time strategy/FPS multiplayer game, and is played by moving and shooting. Players have restricted vision as shown in Figure 6. A player's visible zone is the area to their front and is stopped by solid objects. Players can only see opponents who are in their visible zones.

Using our framework, we produce the model $\mathcal{G}_{TankWar}$, which describes wall-hacking and triggerbotting behaviour. Running $\mathcal{A}$, we obtain detector $\mathcal{D}_{TankWar}$, which is intended to detect the two cheats, as shown below.

Procedure $D_{TankWar}$

```
1 begin
      input  : player, a game client.
      input  : tr, a sequence of server-side game state transitions.
      output: ratedsr_wallhack, ratedsr_triggerbot
2     ratedsr_wallhack = rateDst(tr, player, G.C_wallhack)
3     ratedsr_triggerbot = rateDst(tr, player, G.C_triggerbot)
4 end
```

The detector $D_{TankWar}$ is equipped with client machines $\mathcal{G}_{TankWar}.C_{wallhack}$ and $\mathcal{G}_{TankWar}.C_{triggerbot}$. To examine $\mathcal{D}_{TankWar}$, a strategy is designed as shown in Figure 7. Besides the game server and game clients, there are three more components: a cheat detector, a client monitor and an assessment module. These three components never return data to both server and clients and thus have no influence on the game experience. The client probe is a component independent from the detector. It is made only for validating the performance of the detector by recording each client's behaviour since we have full control of the clients for the experiment.

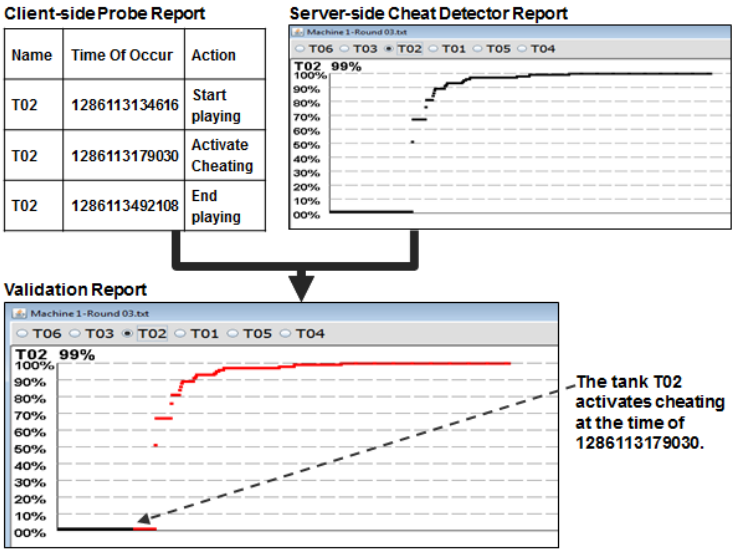

Fig. 8. Example of validation output

The top-right diagram of Figure 8 is an example of the detector's output, presenting the matching rates in a game for a particular player. The x-axis is a timeline and the y-axis shows matching rates. The example detector report describes that the client T02 remained at a low rate at the beginning and increased abruptly to about 50% then gradually rose to about 90% in the end. The top-left of Figure 8 is from the client probe, consisting of three columns: user name, time and action. It reveals when and which clients activated a cheating play mode. The probe report presents that player T02 joined the game at 0s and played in a fair play mode; T02 activated cheating play mode at 45s and finished at 358s. Assessment of the detector's performance is by comparison of the reports from the client probes and the detector.

This validation involves a number of game trials: 32 games were monitored, with 16 for wallhacking and 16 for triggerbotting with five players from a pool of seven.

In most trials, a cheat finished at a rate twice or more as high as a legitimate player. There were only four exceptions, and three of them were accounted for by limitation of data (the players lost the game too quickly) and only one is a true detection failure. The success detection rate is 28 out of 32. We note that the change in detection rate over time was sometimes interesting. For example, some wallhacking detection reports show a rapid rate increase shortly after the beginning. It can be envisaged that cheat detection would perform well when game designers and developers use their knowledge about cheating to describe problems in commercial games.

5 Related Work

Recent research has proposed some novel techniques of cheat detection that do not rely on knowledge about specific game vulnerabilities as many commercial solutions do. They consider cheats by looking at particular measurements. A work based on probability theory was proposed by Chapel et al. [10]. It proposes two complementary frameworks based on the use of the law of large numbers and the Bradley-Terry model [6], which calculate statistical indexes that suggest a player is cheating. The two frameworks are both based on the assumption that each player can be assigned a rank which determines the probability of the outcomes of their games, and determines cheating by observing the difference from resulting ranks and expected ranks. Another novel cheat detection design was proposed by Laurens et al. [17], which we have discussed in previous sections. The design statistically analyses server-side observable behaviour for indications of cheating. For example, to prevent wallhacking, the system collects data (e.g., player view and game world geometry) and then transforms the data to a measure of cheating based on, e.g., frequencies of behind-wall sight vector, distance between players and walls). Subsequently, the measurements are used by statistical algorithms to determine the probability of wallhacking.

Our work is inspired by them and has its own features. It does not rely on any particular measurement (e.g., rank, sight vector, distance). It uses behaviour models to investigate behavioural characteristics and uses the comparison of possible vs. observed behaviours to detect cheats. In common with the methods described above, its performance is resistant to fast-changing cheat implementations due to being a server-side detector.

There are other approaches. For example, it is proposed in [12] that high-level game rules can be described in temporal logic and used to verify the properties of game players at the run-time. The main difference is that our cheat detection does not reply on either rule-enforcement or rule-violation. But, it is done by simulating cheating at instant observation and matching players' behaviours with the simulation result at each instant and calculating to which extent the players are cheating during a reasonable period. In [18], a client patching mechanism is introduced that increases the difficulty of being identified and broken by hackers. The main difference is that our work uses formally-specified simulations of cheaters to detect cheating and does not need patches on game clients.

Moreover, our work has some similarity to *intrusion detection systems* (IDSes). These are primarily focused on identifying possible incidents, which are violations or imminent threats of violation of system security policies or acceptable use policies [16]. A typical IDS records observed events, and (1) matches them with some event patterns (*signatures*) corresponding to known security threats, (2) examines whether or not there are anomalous events using definitions of normal events (*profiles*) or (3) identifies unexpected sequences of events by comparing predetermined profiles of generally accepted definitions of benign protocol activity for each protocol state against the observed events.

The common feature of IDSes and this work is that they all use clients' footprints (or "server-side observable behaviour") on a server or the network to

match certain 'characteristics' and calculate the indication of a threat. Typical IDSes present the characteristics (e.g., *threat signature*, *anomalous profile*, definitions of benign protocol activity) in some form of pattern or description language and match observed behaviours against those patterns. This framework also uses a language —in this case, Event-B— to describe particular cheating behaviours. Our base model $\mathcal{M}$ does not contain any fixed cheating behaviour patterns; it has no direct concern with cheating behaviours and acts as the top abstraction. The validity of $\mathcal{G}$ and the efficiency of the resulting cheat detectors mechanically produced by $\mathcal{A}$ using $\mathcal{G}$ is determined by the quality of the cheating descriptions that are incorporated in $\mathcal{G}$. Similarly, IDSes are only as good as the signatures, profiles or definitions of benign activity.

6 Conclusions

We have demonstrated that we can use refinement in Event-B to describe some cheating behaviours in games. The resulting machines can be used to produce cheat detectors that we argue are more resistant to changing implementations of cheats. We have demonstrated the credibility of describing cheats via formal specification via experiment. This experiment tests both that we can describe the cheats in this fashion and that the resulting detector is accurate.

One might notice that this work shows a different concern from typical work in formal methods. One aim is to bring the precision of formal methods to the description of cheating behaviours. However we have not concerned ourselves with the development of games; we are not, for example, attempting to formally derive or prove the implementation of a game. As a result, it is out of scope for us to address classical issues such as deadlock, fairness, liveness and inconsistency of the games. More detailed designs may be amenable to such analysis depending on the tool support in each instance. Moreover, we are not aware of any significant game designed using such formal specification or analysis.

The major advantage of this work is that it allows game developers to proactively protect their games instead of defending passively against cheat techniques. There is no necessity of capturing behaviour in a temporally adjacent manner and/or at the same interval: for a long-running game, our detectors can randomly sample behaviour several times during a period (e.g., one or two hours), and gradually generate cheating references.

However, we must be able to describe the cheating behaviour in question in Event-B. Essentially, we are trying to describe cheats at a very abstract level such that the implementation of the cheat is inconsequential. Discovery of cheating behaviour is itself an interesting question which we have not attempted to answer here. Automatic construction of these descriptions would require some description of the rules of a game itself; thus these rules would require formal specification.

One limitation is that this work cannot efficiently detect cheats that lead only to trivial behavioural difference between fair players and cheaters. For example, some games allow virtual gifts (e.g., weapons) exchanged between players.

When people abuse this feature and illegally trade, the buyers would obtain an unbalanced (unfair) power against the time they spent. Our detector might not work for this cheat since there might not be a behavioural difference between the buyers and fair players when they exchange their avatars.

Another limitation is on game architecture. The base model $\mathcal{M}$ is an abstraction of client/server architecture games, and the behaviour model $\mathcal{G}$ must be derived from it. However, this does not suit all game architectures, some of which may have strong peer-to-peer components.

This work leaves open many possible future improvements.

- Introduce a set of base models, $\mathcal{M}_{1..n}$ to allow for a broader range of game types.
- Developing easier ways to produce the predicates that describe cheating behaviour. A further development would be to automatically identify possible predicates describing cheating; but this itself would require some description of the intent and rules of any particular game. We speculate that particular patterns may arise often enough that they could be described generically.
- Extend our framework to be able to identify game players from their behaviour. This would work even if they use different user names. This can prevent people from cultivating avatars or farming games for buyers, or from changing IP address to avoid countermeasures such as IP blacklisting.

Although we have used this framework only for the classification of game users into fair-player and cheating-player groups in this paper, its application can feasibly be extended to the classification of users in other multiuser systems, such as a social-networking system. $\mathcal{A}$ could be adapted from being a creator of cheat detectors to a creator of user classifiers. A user classifier could put social network users into a variety of categories (e.g., a movie fan, a sport fan, etc.) and even sub-categories (e.g., an action movie fan, a bicycle sport fan, etc.) by calculating server visible behaviours. Thus, it might facilitate solutions for delivering relevant content to the users who are most likely to be interested.

Acknowledgements

This work was undertaken during the first author's PhD studies at Teesside University, and was funded by a research doctoral scholarship from that university. We thank the anonymous reviewers for their detailed and constructive remarks.

References

1. Jean-Raymond Abrial. *The B-book: Assigning Programs to Meanings.* Cambridge University Press, New York, NY, USA, 1996.
2. Jean-Raymond Abrial. *A System Development Process with Event-B and the Rodin Platform.* 2007.
3. Jean-Raymond Abrial. *Modelling in Event-B: System and Software Engineering.* Cambridge University Press, 1 edition, 2010.

4. Jean-Raymond Abrial and Laurent Voisin. Rodin deliverable 3.2 Event-B language. 2005.
5. Mark Bell. Toward a definition of "virtual worlds". *Journal of Virtual World Research*, 1(1), 2008.
6. Ralph Allan Bradley and Milton E. Terry. *Rank Analysis of Incomplete Block Designs: I. The Method of Paired Comparisons*, volume 39. Biometrika, 1952.
7. Phillip J. Brooke, Richard F. Paige, John A. Clark, and Susan Stepney. Playing the game: cheating, loopholes, and virtual identity. *SIGCAS Comput. Soc.*, 34(2), September 2004.
8. Michael Butler. Decomposition structures for Event-B. In *Integrated Formal Methods iFM2009, Springer, LNCS 5423*, volume LNCS. Springer, February 2009.
9. Ana Cavalcanti and Jim Woodcock. The semantics of *circus*. In D. Bert, J. P. Bowen, M. C. Henson, and K. Robinson, editors, *ZB 2002: Formal Specification and Development in Z and B*, volume 2272 of *Lecture Notes in Computer Science*, pages 184—-203. Springer-Verlag, 2002.
10. Laetitia Chapel, Dmitri Botvich, and David Malone. Probabilistic approaches to cheating detection in online games. In *Computational Intelligence and Games*. IEEE Symposium on, 2010.
11. Stefano de Paoli and Aphra Kerr. We will always be one step ahead of them: A case study on the economy of cheating in MMORPGs. *Journal of Virtual Worlds Research*, 2(4), 2010.
12. Margaret DeLap, Björn Knutsson, Honghui Lu, Oleg Sokolsky, Usa Sammapun, Insup Lee, and Christos Tsarouchis. Is runtime verification applicable to cheat detection? In *Proceedings of 3rd ACM SIGCOMM workshop on Network and system support for games*, NetGames '04, pages 134–138, New York, NY, USA, 2004. ACM.
13. DMW. Available at `http://www.dmwworld.com/`, 2002.
14. INCA Internet. Available at `http://www.inca.co.kr/`, 2000.
15. Hyun-Jin Choi Jian Xin, Jeff Yan. Security issues in online games. *The Electronic Library*, 20, 2002.
16. Scarfone Karen and Mell Peter. *Guide to Intrusion Detection and Prevention Systems (IDPS)*. Computer Security Resource Center, 2009.
17. Peter Laurens, Richard F. Paige, Phillip J. Brooke, and Howard Chivers. A novel approach to the detection of cheating in multiplayer online games. In *ICECCS '07: Proceedings of the 12th IEEE International Conference on Engineering Complex Computer Systems*, pages 97–106, Washington, DC, USA, 2007. IEEE Computer Society.
18. Christian Mönch, Gisle Grimen, and Roger Midtstraum. Protecting online games against cheating. In *Proceedings of 5th ACM SIGCOMM workshop on Network and system support for games*, NetGames '06, New York, NY, USA, 2006. ACM.
19. Ralph Schroeder. Defining virtual worlds and virtual environments. *Journal of Virtual Worlds Research*, 1(1):1–3, 2008.
20. HaiYun Tian. *Formal Derivation of Behaviour-based Cheat Detectors for Multiplayer Games*, PhD thesis. School of Computing, Teesside University, 2012.
21. Valve Corporation. Available at `http://www.valvesoftware.com/`, 2002.
22. Jim Woodcock and Jim Davies. *Using Z - Specification, Refinement, and Proof*. Prentice Hall, 1996.
23. Jeff Yan and Brian Randell. A systematic classification of cheating in online games. In *NetGames '05: Proceedings of 4th ACM SIGCOMM workshop on Network and system support for games*, pages 1–9, New York, NY, USA, 2005. ACM.