Game Feature Validation of a Real-Time Game Space with an eXtended Classifier System

Damijan Novak
Faculty of Electrical Engineering and Computer Science
University of Maribor
Maribor, Slovenia
damijan.novak@um.si

Iztok Fister Jr.
Faculty of Electrical Engineering and Computer Science
University of Maribor
Maribor, Slovenia
iztok.fister1@um.si

2021 IEEE 25th International Conference on Intelligent Engineering Systems (INES) | 978-1-6654-4499-6/21/$31.00 ©2021 IEEE | DOI: 10.1109/INES52918.2021.9512921

***Abstract*— The objective of this paper is the proposal of a new approach for the game feature validation of a game space with the eXtended Classifier System (XCS) algorithm. For initial "proof-of-concept" evaluation we used the game space of the Tic-Tac-Toe game, which was placed in a context of real-time characteristics. Evaluation was done with the XCS algorithm without pre-processing internal knowledge programmed (online mode). Evaluation data results were acquired under real-time constraints. No-loss-strategy was implemented for the game, with various scenarios tested to find out if the algorithm has the capability of finding invalid game feature design flaws. Results indicated that the XCS algorithm is able to deliver stable validation of game feature testing results, can be a powerful tool and, therefore, worthy of further research in the Gaming domain. Confirmation of this core concept testing with this type of algorithm (and similar ones) is necessary, since it paves the way for further research in more complex game environments, where the dimension size, the number of aspects, game states and game actions rises intensely.**

Keywords*— *Artificial Intelligence in Games, Game Feature, Game Feature Validation, Game Space, No-Loss-Strategy, XCS.

I. INTRODUCTION

Development of a new game (space) is a very complex process. It depends on three factors: The time (needed to complete the project), human resources and the scope of the project [1]. The time needed to complete the project is often fixed (i.e. the game or product must be released in the time promised to the gamers). Scope also cannot be changed easily (it is set in advance by a Game Design Document (GDC), which specifies core gameplay, game elements, necessary Game Features (GF), etc. [2]). The resources component is, therefore, a variable that can be affected in two ways. Either we raise the number of developers working on the project, or we extend their working hours. As a consequence, the price of the project may be higher, and milestones coming to an end will increase developers` workload and stress (crunch time), which brings a negative influence [3].

Creating a new game space is quite a challenge, because testing it is, due to the complexity of game spaces, certainly not an easy task [4]. Also, the automatic game testing is a largely untouched niche, and it still relies mainly on manual testing [5], and manual testing is usually inefficient and subjective [6]. Fortunately, we can observe increasing research interest in the domain of Automated Game Testing (e.g. game agents based on AI methods, like Sarsa and MCTS, which were transformed for the game testing purposes [7], or modern frameworks for autonomous video game playing [8]).

In this paper, we tackle the time-consuming game space validation problem with a "proof-of-concept" utilization of automated validation of GF of a Tic-tac-toe game space through the usage of an eXtended Classifier System (XCS) algorithm. XCS is a Reinforcement Learning (RL) type of an algorithm [9]. We utilize the XCS algorithm for indirect checking if all the necessary GF work is as specified and intended, while the algorithm is learning to play the game across all the game paths. We also focus on obtaining results in real-time. By real-time we are not referring to hard, but to the soft real-time (games can be considered as soft real-time applications [10]), where an agreed foreseeable time value is set. In soft real-time, if the result is received after the agreed time limit, the result is still useful, only its quality can deteriorate with passing time.

II. GAME FEATURE

In the study of Heintz and Law [11], they defined GF as the following: "*Game features is a generic term used to refer to differences and similarities between games, which is further refined by the terms "game elements" and "game attributes"...*", and in the study of Sicart [12] the Game Mechanics (GM) were defined as: "*Game mechanics are methods invoked by agents for interacting with the game world*". Both definitions become connected or linked inextricably, after the implementation phase is complete (i.e. the cycle of their beginning definement in GDC until their final implementation is complete). GM connects with each other the game rules, the game objects and their properties (included is the correct usage of the game object), and the game environment, so that immersive game scenarios are created where a player can progress through the game in a predictive way [13]. In such a scenario, game feature becomes the holder, or reflects the specific characteristic of the game [14] (e.g. A game object wall for which the GM defined indestructibility becomes the important game feature placed in the story of a prison break). GF can be divided in a group of functional and a group of non-functional requirements [15]. Functional requirements are those that define the basic system behavior, like what the system can and cannot do [16]. Non-functional requirements, however, specify how, or in which way, the system should do that (e.g. what the emotional reaction should be like when playing a game, or when the specific game feature is activated) [17]. With this work, we focus on the group of functional requirements, as they are measurable (the exact output is provided / expected for a given input).

III. TIC-TAC-TOE GAME SPACE

The basic Tic-Tac-Toe game is played on a map consisting of 9 (3x3) cells, where each cell can be either empty, or occupied by a cross/circle. In every non-final game state, the player whose turn it is, can execute an action of positioning his/her game element (cross or circle) into an empty cell. Fig. 1 presents one such game state of the game space Tic-tac-toe, with two elements already positioned and with seven empty cells (2, 3, 4, 5, 6, 7 and 8).

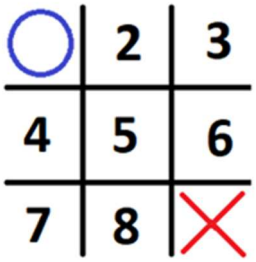

Fig. 1. One example of the Tic-Tac-Toe game state.

Let us define a full search space (also known as state-space [18]) as a set S, which holds all the states of all possible configurations of the game map. Set S therefore holds 3^9 (19.683) states. If we proceed and include the game elements into the sets of states and actions, the states and actions form syntactical and semantical connections, on the basis of which the search space can be reduced. This means we are only left with the legal states and legal actions (i.e. only the actions that are allowed to be executed in a specific state [19]). From the whole state-space of the game Tic-Tac-Toe (19.683), we are left with 5.478 legal states, after the illegal ones have been excluded (e.g. six crosses present on the game board, but no circles).

The legal (reduced) game search space can now be defined as set S, which contains only the legal layouts of the game board that can be achieved from the starting position of the game [20]. Also, for the set of actions A in such a space, the following applies: If, over a state $S_i \in S, \forall i > 0$, we can only perform the legal moves (actions) A_{Si}, then A is defined as $A = \{A_{S1}, A_{S2}, ..., A_{Sn}\}$, where n is the number of all the legal states contained in the set S. Sometimes such a reduced game search space is connected closely to the term decision complexity, where traversing through the search space are made (with close attention to the game definitions/rules) in search of the optimal strategies [21], but the whole thing can be subjective, on account of the different game rules (players can play the same types of board games, but by following different variations of the game rules).

The full (and sometimes even reduced) search spaces can be overwhelming from the available computer resources standing, so an understanding of the workings of the core "proof-of-concept" research examples, such as we are trying to achieve with this article of game feature validation with XCS, is in our opinion, surely needed. The low number of legal games and useful states, on the other hand, make it possible to represent the whole game in the memory as a game tree without any restrictions (a full game tree of Tic-Tac-Toe inside the memory [22][23]). Such a game tree can be built, even with the basic algorithms (e.g. Minimax algorithm), and, during creation, we can make optimizations regarding the lower computer resources usage, by only focusing on the optimal (or useful) game strategies [24]. A no-loss strategy can be identified out of the useful set of the optimal game strategies. If both players choose to play with a no-loss strategy, the game will always end in a tie.

IV. eXTENDED CLASSIFIER SYSTEM

A. General description

The XCS algorithms belong to the group of Learning Classifier Systems (LCS), which are rule-based systems [25]. Rules are kept in a population set, which (as a whole set) represents the candidate solution to the problem. In the LCS domain the rules (called classifiers) are presented in the form of "IF condition THEN action". Condition is a vector, and it represents one form of the environment, while the action part stands for the action that the classifier is propagating. The vector of the condition can be a mix of bits (0 and 1) and #'s (don't care, or not relevant bits). The # symbol is introduced with the purpose of conditions being able to generalize (e.g. the condition 1#0 represents 110, as well as 100). LCS's learn and evolve new classifiers through interaction with the environment (i.e. by executing actions for which the indication of the value of the action, called reward, is received). For the purpose of interaction with the environment, the LCS algorithms usually utilize the help in the form of an environment programme (acting as an interface between the LCS algorithm and the environment), and reinforcement programme (to provide the LCS algorithm with a suitable reward while interacting with the environment). In the basic form, the environment programme provides the LCS algorithm with input information (also called sensory information), and the LCS algorithm outputs an action which the environment programme executes in the environment. The reinforcement programme's role is to use an evaluation function for scoring the environment (game state).

For this article we chose the XCS algorithm, because the literature shows [26] that it can learn a state-value function over the complete state-action space through efficient generalizations. We implemented the XCS algorithm by following the detailed specifications presented in the work of Butz and Wilson [27]. The XCS classifiers also hold additional parameters, which are, otherwise, not used in all the LCS algorithms, but they contribute importantly to the XCS's efficient generalizations. These additional parameters are: Prediction estimate (payoff that we can expect if the condition of the classifier will match the environment input), prediction error (estimates the errors made in the prediction) and experience (a counter of how many times the classifier has belonged to the set of chosen (propagating) actions). Also, inside the XCS classifier, we have an important parameter of fitness. Fitness tells us how accurate the prediction estimate is, since we are not just interested in the prediction estimate, but also in how accurate this prediction is.

B. Syntactic and Semantic Validation of Classifiers

If we want to connect the XCS algorithm successfully to the game space of Tic-Tac-Toe, we need to have the syntactically and semantically valid classifiers [28]. The validation is needed, because otherwise an XCS cycle could get stuck, or wouldn't generalize efficiently. The invalidness of classifiers can occur as a result of the random classifier creation (e.g. random creation of the population set), while executing classifier covering (which can occur during match set creation), or during the execution of the genetic algorithm (crossover operations).

With the syntactic validation we therefore ensure that the condition of the classifier (represented as one variation of the game state), and the action, which will be executed upon the game state, are compatible. Namely, it checks two things. First, if the cell, which is a part of the game state represented in the condition and on which the element is going to be placed, is free, and second, if the number of placed crosses on the board does not differ by more than one compared to the number of placed circles (this would violate the rules of the game). With semantic validation we make sure if the chosen action of an XCS algorithm can be executed on the live game

state (e.g. we test if we are putting a symbol on the empty cell). Meaning, the semantic validation is not performed on the condition, but always on the live game state (the state of the environment). If the classifier during either validation doesn't check out, such a classifier is disregarded for future use.

V. Validation component

A. Design of the Validation Component

Game agents are vital parts of many games [29], with the main assumption of their function being intelligent and rational [30], so they can usually be seen in the role of intelligently controlled game characters [31]. Such intelligent agents are designed to be good performers while playing the game, so we would like to use them for GF validation as well. It is advised that, for validation purposes, a game agent has good implementation design of interfaces for inputs (game states) and outputs (e.g. actions to play with), as well as being built in a component fashion. This way the game agent is connected easily to the game environment. Also, we can observe clearly what a game agent received as input, and what its actions were in regard to that input.

To validate GF with a game agent, we need to adapt its game-playing mechanics for the play-testing purpose. In our case, we created and used the game constituent part we call a non-invasive component. The component`s purpose is to observe and measure the game agents behavior (when it is involved actively with the environment), and what consequences its game actions have on the game state.

The non-invasive component is summed up in the following steps:

1. Acquiring the values of a measure: The component needs the measured data about the agent`s workings (actions that were performed) and information about the game state (state of the game map and positions of all the game objects on it – crosses and circles).
2. Executing of the internal method designed to validate the specific game feature: From the previous step the method receives the measured data as input parameters. The method is the center piece and crucial part of the component. It follows the agent`s behavior iteratively, and uses the acquired information about the impact of the executed actions on the game state, with the purpose of overwatching if there was a breach of validity of the game feature. Iteration (transition of one game state to another) is executed after the execution of actions by both players. The breach of validity is tested by the conditional statement.
3. Design and execution of the conditional statement: (With every iteration of the game state) the conditional statement checks if the new game state is breaking the design (purpose) of the game feature (e.g. part of the map which should always be empty is now occupied). Note: "With every iteration of the game state" was put into squares, because not all the GF require to be tested with every game state, but it depends mainly on what the game functionality specifies or requires explicitly. If the game feature specifies that the player must not lose a game in a specific scenario, it is only logical that we test the game feature with a conditional statement after the game is over (on the last game state).
4. Execution of the conditional statement`s condition: The conditional statement is the holder of the condition which is compared against the measured values. The condition must always return false, which states that the game feature is valid, because the condition testing for invalidity was not successful (e.g. the conditional statement holds the condition which is responsible for checking if the specific part of the map is empty – the measured value is the state of the map). If the condition is true, the game feature is, therefore, invalid (the game developer will have to correct the problem).

We refer to the XCS algorithm when connected to the game for the purpose of playing, and when it has been overseen by a component non-intrusively, as a play-testing agent.

B. The conditional statement: The make-up of the condition

Depending on the type of measured values, the design of the condition can be performed in a simple or in a complex way (e.g. by categorizing the individual enemy units into groups and checking the points of interest). In a simple way we use the measured values which are of a basic data type, and where the comparison against the condition can be direct (e.g. the player had won: true/false). The processing of not just basic data types is performed in a complex way, and also with the complex data types (e.g. dynamic list of game objects). During the condition checking, we must only be cautious that the checking of the condition isn't too time-consuming (e.g. recursive calls), and that the available execution time would be surpassed. Time slice, which is the maximum available time between the transition of one game state to another, could be surpassed, and the processing of the data needed for the condition is not necessarily finished.

VI. Experiment

In the experiment we used the Game Feature, which is presented in the form of a fully-operational (i.e. without known bugs) No-loss-strategy method. Then, deliberate bugs (errors) were inserted into this method. The purpose was to test the XCS algorithm to see if it discovered successfully, (through gameplay), that the method was not operating to its no-loss specifications when the bugs were present.

A. Experimental environment and its settings

Hardware and software environment

The experiment was carried out on an i7-9700 CPU computer @ 3 GHz (turbo: 4.7) GHz, 8 cores, 32 GB RAM, OS Windows 10 Pro and Java version 13.0.2 with Integrated Development Environment (IDE) 2020.2.1 (Community Edition) programming editor.

Experiment design

During the experiment we used the game space of Tic-Tac-Toe, which was designed to operate in real-time. Real-time operation was achieved by not implementing any delays (hard-coded time slices) between the execution of two actions. So, as soon as the first player executes an action, it is the second player`s turn to make its own (and so on, until the game is finished). The first player is controlled by the XCS algorithm, and the second player is implemented with a no-loss-strategy method (Appendix A). Method operation was as follows: For the chosen symbol (cross or circle) the method outputs the position of the cell into which the player, who is

the holder of the symbol, should position his/her element. If we play against a game agent who uses this method, we will never win. Meaning, our no-loss strategy is comprised of many optimal sub-strategies, which together form a no-loss strategy (e.g. if the game has just started, we position the first element in the center of the game map).

The experiment consisted of two parts. In the first part, the game space of Tic-Tac-Toe was comprised of only the valid GF, acting as a test for XCS to confirm the validness of the no-loss-strategy. When playing against such an opponent, the XCS algorithm must not win. In the second part, intentional bugs were introduced into the GF (Table I), making them invalid, which acted as a test of how successful the XCS was in finding those bugs. Both experiments were executing blocks of games, with one block consisting of one million full-game playouts. The first part of the experiment was run by executing one block of games, while the second part was run on two hundred blocks for each of the bugs intentionally included into the GF.

TABLE I. BUGS INSERTED IN THE NO-LOSS STRATEGY METHOD.

ID of the bug	*Line numbers missing from the method (starting and ending positions)*	*Short description of a bug*
1.	6-8	Opportunity of putting a marker in position five is not taken, if an opponent already has one marker on the board and it is not occupying number five.
2.	9-12	Opportunity of a certain victory in the next move is not taken.
3.	13-16	Opportunity of preventing a certain victory for the opponent is not taken.
4.	18-29	Opportunity to prevent the opponent creating a scissor in the next move is not taken.
5.	30-33	Opportunity to prevent the opponent from any future scissor tactics (during the next moves) is not taken.
6.	34-38	Same description as before.
7.	40-43	Opportunity to block corners (lowering the search space and possibility of wins for the opponent) if there is only one marker on the map that is not taken.
8.	44-62	Similar description as before, but independent of the number of markers already present on the map.

The purpose of repeating the execution of blocks two hundred times was to test the performance of the XCS algorithm with greater probability of the results being reliable. In the XCS algorithm, due to the use of random values (e.g. random placement of the initial population of classifiers), the result for each block is not always the same. So, if we executed only a single block, we risked acquiring the exceptionally (non)successful block measurement (as a result of the randomness involved in the XCS algorithm), which wouldn't be representative. So, with higher block repeats, we could provide a range of performance evaluation in the form of an interval, where the limits of the interval were determined by the most successful and least successful blocks.

After the execution of each block, all the internal values of the XCS algorithm, as well as its supporting programmes, were reset to the starting values (reinitialization of all the values). This ensured that the knowledge stored in the population of the XCS algorithm was not transferred between the blocks, and that the execution of the blocks was independent of each other.

Data acquired during the experiment

During the experiment, the following data were acquired (per each block):

- the consecutive number of the game in which the irregularity of the game functionality was first discovered,
- the number of all confirmations of the invalid game feature, and
- the time needed for completion of each block (the time measurements provided us with information if it was possible to validate the game space in real-time).

The upper bound of time, where the real-time measurement during one block execution is still acceptable, was set to a reasonable one minute [32].

Parameter settings of the XCS algorithm

The parameter setting of the XCS algorithm have to be set before the XCS cycle begins, and they stay static during the execution phase. We set the following values: NP = 100 (population size), α = 0.1 (learning rate for updating ρ), β = 0.2 (learning rate for p, ε and f), γ = 0.71(discount factor of the reward), δ = 0.1 (below this value, the fitness of a classifier may be considered in its probability of deletion, so, in other words, it specifies the fraction of the mean fitness in [P]), $\varepsilon 0$ = 10 (the error below which classifiers are considered to have equal accuracy), θGA = 25 (GA threshold), θdel = 20 (deletion threshold), ν = 5 (power parameter), χ = 0.5 (crossover probabilities), μ = 0.01 (mutation probability), θsub = 20 (subsumption threshold), $P\#$ = 0.33 (probability of using a # in one attribute in a condition when covering), $pI = \varepsilon I = fI = 0$ (used as initial values in new classifiers), $pexp$ = 0.02 (exploration probability), [A] subsumption and GA subsumption were set to false.

The values were set during the initial pre-testing of the algorithm, with the goal of XCS having the optimal performance through hand-settings, but, for now, they weren't optimized by any of the advanced algorithmic methods [33].

Map coding Table for the Tic-Tac-Toe

The condition of classifier and sensory information (input of the environment) must be of the same length and of the same representation to be comparable. They are both built upon the representation of the game map, meaning the encoding is done by representing each state of the cell of the map in a binary form (Table II). A cell is represented with two bits. The two-dimensional map of Tic-Tac-Toe is, after completed encoding, presented as a binary vector of the size 18 (9 cells x 2 bits per cell).

TABLE II. MAP CODING TABLE FOR THE TIC-TAC-TOE.

State of the cell	*Representation in bits (2 bits per cell)*
Cell is empty (not occupied)	0 0
Cell is occupied by a cross (X)	0 1
Cell is occupied by a circle (0)	1 0
Non-relevant cell (set with don`t care flag in the classifier condition)	1 1

No-loss strategy game feature and its settings for the experiment

To test the validity of the Game Feature, implemented as a No-Loss-Strategy method in Appendix A, we must first define the conditional statement and its condition (measured values are the end state of the game – defeat or tie). They were as follows:

if resultOfTheGame **not** defeat **and** resultOfTheGame **not** tie **then**

// game feature is not valid

end if

If the result of the game is not defeat and, at the same time, it isn't a tie, the game for the player who plays against the NoLossStrategy method resulted in a win (meaning the condition is true, and game feature therefore being invalid). We pre-tested the implementation of the NoLossStrategy method with 100 million playouts by the XCS algorithm, and additionally with one million playouts by a basic MCTS (UCT) reinforcement learning algorithm [34]. None of the playouts resulted in a win (game feature was always valid).

VII. Results and Discussion

We can confirm that the XCS algorithm was successful in finding invalid GF caused by bugs in the allotted time, and every bug had multiple confirmations per each block, as shown in Table III.

TABLE III. Results of the Experiment.

ID of the bug	*Consecutive number of the game where XCS first confirmed invalid game feature [min. cons. num. of most succ. block, max. cons. num. of least succ. block] (The lower the values the better.)*	*The number of all confirmations of the invalid game feature in the block [min. num. conf. of least succ. block, max. num. conf. of most. succ. block] (The higher the values the better.)*	*Time needed to execute the block in seconds [min. time. of most. succ. block, max. time. of least succ. block] (The lower the values the better.)*
1.	[1, 1784]	[38413, 60621]	[32.651, 33.893]
2.	[45, 109330]	[135, 1200]	[29.737, 31.706]
3.	[1, 2]	[920156, 930109]	[23.963, 25.699]
4.	[1259, 504222]	[4, 339]	[30.819, 33.267]
5.	[20, 17783]	[1329, 3240]	[30.371, 33.191]
6.	[1, 91558]	[65, 154]	[30.154, 33.132]
7.	[24, 22855]	[254, 470]	[29.945, 31.553]
8.	[11, 13322]	[3701, 7616]	[31.256, 32.602]

The most successful Game Feature validation was the bug with the ID 3. The invalidity of the game feature always happened on the first or at the second consecutive game run at the latest; it reached the highest number of confirmations per single block (interval being [920156, 930109]) and in the lowest amount of time. The least successful (but still successful) game feature validation was at the bug with the ID 4. Here, the lowest number of confirmations of the invalid game feature was four, with the greatest interval range ([1259, 504222]. Maximum recorded time during the experiment was 33.893 s, which was well under the limit of one minute of soft real-time per block.

The consecutive numbers and the numbers of all confirmations were quite varied between the different inserted bugs. This was somewhat expected, because the bugs were connected closely to the search-space. Some bugs are more common to be found, and some occur rarely during the game, meaning only when an XCS algorithm is traversing through the niche parts of the game (i.e. only in a few sub-search-spaces), and that can sometimes take many tries. Nevertheless, the XCS algorithm identified the bugs successfully, regardless of them occurring in common or niche search space paths, and it also confirmed them multiple times per each block. This confirms our beliefs that XCS is a suitable candidate for testing and research in more complex game spaces and their GS.

VIII. Conclusion

In this paper, we proposed the XCS algorithm for the task of Game Feature validation of the real-time game space of Tic-Tac-Toe. The important game feature of No-loss-strategy was implemented inside this game space. Then, various bugs were inserted into the No-Loss-Strategy method, and tested by the XCS algorithm, which was under the supervision of the non-invasive validation component. Experiments revealed, that XCS identified all the invalid GF successfully, and it did so in the real-time that was defined. This serves us as a proof-of-concept that the XCS algorithm can be a capable tool for validation of game spaces, and that it can act in online mode (without prior learning knowledge of the environment).

Therefore, our future work will focus on game genres which enclose game spaces of higher computational complexity [35]. Specifically, we want to test our methodology of Game Feature validation with the XCS algorithm on game spaces which include maps of higher resolution (e.g. 32 x 32 cells), game actions with more diversity (i.e. durative game actions and simultaneous moves), and with more than one type of game unit available (i.e. in Tic-Tac-Toe the only choice for the player is to "operate" one game unit, cross or circle).

The durative game actions and simultaneous moves are some of the main differences between classical board games and complex game genres (e.g. Real-Time Strategy games) [36]. The XCS algorithm in its current state only outputs one best action of its choosing, so it does not support choosing of multiple actions simultaneously. Therefore, incorporating game actions with higher diversity in the XCS algorithm will be challenging.

We envision creating a game agent, which has its internal structure designed for the operation of multiple XCS algorithms, and will be capable of handling all the game units on the map simultaneously, each with its own distinct actions. It will also be necessary for a game agent to incorporate more complex syntactic and semantic validations of classifiers, as well as a game state evaluation and reward system of higher complexity.

Combining the power of adaptive algorithms, such as an XCS algorithm, for the purpose of the playtesting of GF, could eventually help with many tasks that are currently manual labor intensive in the game design. This way we could possibly open ways for not only faster and cheaper creations of games, but also for the possibility of creating different

varieties of the same game (i.e. validating the same GF with different criteria), with a lower number of bugs and higher user gameplay satisfaction.

ACKNOWLEDGMENT

The authors acknowledge the financial support from the Slovenian Research Agency (Research Core Funding No. P2-0057).

REFERENCES

[1] C. Buhl, and F. Gareeboo, "Automated testing: a key factor for success in video game development. Case study and lessons learned," in Proc. PNSQC 2012, pp. 1-15, 2012.

[2] M. G. Salazar, H. A. Mitre, C. L. Olalde, and J. L. G. Sánchez, "Proposal of Game Design Document from software engineering requirements perspective," in Conference on CGAMES 2012, pp. 81-85, 2012.

[3] H. Edholm, M. Lidstrom, J. P. Steghöfer, and H. Burden, "Crunch Time: The Reasons and Effects of Unpaid Overtime in the Games Industry," in ICSE-SEIP 2017, pp. 43-52, 2017.

[4] C. Redavid, and A. Farid, "An overview of game testing techniques," Västerås: sn, 2011.

[5] Y. Zheng, X. Xie, T. Su, L. Ma, J. Hao, Z. Meng, Y. Liu, R. Shen, Y. Chen, and C. Fan, "Wuji: Automatic online combat game testing using evolutionary deep reinforcement learning," in 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), IEEE, pp. 772-784, 2019.

[6] L. K. Chen, Y. H. Chen, S. F. Chang, and S. C. Chang, "A Long/Short-Term Memory Based Automated Testing Model to Quantitatively Evaluate Game Design," Applied Sciences 10(19): 6704, 2020.

[7] S. Ariyurek, A. Betin-Can, and E. Surer, "Automated Video Game Testing Using Synthetic and Human-Like Agents," IEEE Transactions on Games, IEEE, 2019.

[8] J. Pfau, J. D. Smeddinck, and R. Malaka, "Automated game testing with icarus: Intelligent completion of adventure riddles via unsupervised solving," in Extended Abstracts Publication of the Annual Symposium on Computer-Human Interaction in Play, pp. 153-164, 2017.

[9] M. V. Butz, D. E. Goldberg, and P. L. Lanzi, "Gradient descent methods in learning classifier systems: Improving XCS performance in multistep problems," IEEE Transactions on Evolutionary Computation, 9(5), IEEE, pp. 452-473, 2005.

[10] B. M. Zamith, L. Valente, and E. Clua, "Game loop model properties and characteristics on multi-core cpu and gpu games," SBGames 2016, 2016.

[11] S. Heintz, and E. L. C. Law, "Digital educational games: methodologies for evaluating the impact of game type," ACM Transactions on Computer-Human Interaction (TOCHI) 25(2), pp. 1-47, 2018.

[12] M. Sicart, "Defining game mechanics," Game Studies 8(2), n, 2008.

[13] R. J. Mislevy, S. Corrigan, A. Oranje, K. DiCerbo, M. I. Bauer, A. von Davier, and M. John, "Psychometrics and game-based assessment. Technology and testing: Improving educational and psychological measurement," pp. 23-48, 2016.

[14] R. DeRouin-Jessen, "Game on: The impact of game features in computer-based training," 2008.

[15] V. T. Sarinho, G. S. de Azevedo, and F. M. Boaventura, "Askme: A feature-based approach to develop multiplatform quiz games," in 2018 17th Brazilian Symposium on Computer Games and Digital Entertainment, SBGames, IEEE, pp. 389-398, 2018.

[16] L. T. G. Ferreira, "Eye Tracking User Interface," 2020.

[17] C. Alves, G. Ramalho, and A. Damasceno, "Challenges in requirements engineering for mobile games development: The meantime case study," in 15th IEEE International Requirements Engineering Conference (RE 2007), IEEE, pp. 275-280, 2007.

[18] H. J. Van Den Herik, J. W. Uiterwijk, and J. Van Rijswijck, "Games solved: Now and in the future," Artificial Intelligence 134(1-2), pp. 277-311, 2002.

[19] J. Baxter, A. Tridgell, and L. Weaver, "Learning to play chess using temporal differences," Machine Learning 40(3), pp. 243-263, 2000.

[20] L. V. Allis, "Searching for solutions in games and artificial intelligence," Wageningen: Ponsen & Looijen, pp. 21-152, 1994.

[21] C. Xu, Y. Zhao, and J. F. Zhang, "Decision-implementation complexity of cooperative game systems," Science China Information Sciences 60(11): 112201, 2017.

[22] S. D. James, "The effect of simulation bias on action selection in Monte Carlo Tree Search," Doctoral dissertation, 2016.

[23] A. Singh, K. Deep, and A. Nagar, "A "Never-Loose" Strategy to Play the Game of Tic-Tac-Toe," in 2014 ISCMI, IEEE, pp. 1-5, 2014.

[24] S. G. Diez, J. Laforge, and M. Saerens, "Rminimax: An optimally randomized MINIMAX algorithm," IEEE transactions on cybernetics 43(1), pp. 385-393, 2012.

[25] K. Shafi, and H. A. Abbass, "A survey of learning classifier systems in games," IEEE Computational intelligence magazine 12(1), pp. 42-55, IEEE, 2017.

[26] M. V. Butz, D.E. Goldberg, and P.I. Lanzi, "Gradient descent methods in learning classifier systems: improving XCS performance in multistep problems," Illigal Report 2003028, Illinois Genetic Algorithms Laboratory, 2003.

[27] M. V. Butz, and S. W. Wilson, "An algorithmic description of XCS," in International Workshop on Learning Classifier Systems, pp. 253-272, Springer, Berlin, Heidelberg, 2000.

[28] A. S. Vasilyev, "Classifier systems learning in dynamic environment," Scientific proceeding of Riga Technical Unniversity, vol. 5, Datorzinatne, Information technology and management science, 5. sejums, pp. 175-187, 1999.

[29] M. Jacob, S. Devlin, and K. Hofmann, ""It's Unwieldy and It Takes a Lot of Time"—Challenges and Opportunities for Creating Agents in Commercial Games," in Proceedings of the AAAI Conference on AIIDE, vol. 16, no. 1, pp. 88-94, 2020.

[30] S. Kooistra, "Logic in classical and evolutionary games," 2013.

[31] A. Nurhuda, and R., "Andrea Implementation of Decision Tree Algorithm on Game Agent of First Aid Educational Game," in Asian Conference on Intelligent Information and Database Systems, Springer, Cham, pp. 313-322, 2019.

[32] D. Stefanovic, and M. N. Stojanovic, "Computing game strategies," in Conference on Computability in Europe, Springer, Berlin, Heidelberg, pp. 383-392, 2013.

[33] J. S. Bergstra, R. Bardenet, Y. Bengio, and B. Kégl, "Algorithms for hyper-parameter optimization," in Advances in neural information processing systems, pp. 2546-2554, 2011.

[34] L. Kocsis, and C. Szepesvári, "Bandit based monte-carlo planning," in European conference on machine learning, Springer, Berlin, Heidelberg, pp. 282-293, 2006.

[35] G. Synnaeve, and P. Bessiere, "Multiscale Bayesian modeling for RTS games: An application to StarCraft AI," in IEEE Transactions on Computational intelligence and AI in Games, 8(4), pp. 338-350, 2015.

[36] A. Uriarte, and S. Ontanón, "Game-tree search over high-level game states in RTS games," in Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment, 10(1), 2014.

IX. APPENDIX A

Presumption: method has full access to the game state / map

Function noLossStrategy(markerFriendly, markerOponnent):

```
1.  set counter = getCountGameStateNonEmptyCells()
2.  if counter = 0 then
3.     return new Position(5)
4.  end if
5.  set emptyCell = isTheCellEmpty(new Position(5))
    // opponent has a marker on the board, but not in the middle, so middle cell can
    // be taken
6.  if counter = 1 and emptyCell then
7.     return new Position(5)
8.  end if
    // if there is a possibility of a definite victory take advantage of it
9.  set takeAdvantageFriendly = takeAdvantageIfVictory(markerFriendly)
10. if not emptyPosition(takeAdvantageFriendly) then
11.    return takeAdvantageFriendly
12. end if
    // if enemy has definitive win in the next move, it has to be blocked
13. set takeAdvantageEnemy = takeAdvantageIfVictory (markerOponnent)
```

```
14.  if not emptyPosition(takeAdvantageEnemy) then
15.    return takeAdvantageEnemy
16.  end if
17.  if counter = 3 then
18.    if gameState[2] = markerOponnent and gameState[4] = markerOponnent then
19.      return new Position(1)
20.    end if
21.    if gameState[2] = markerOponnent and gameState[6] = markerOponnent then
22.      return new Position(3)
23.    end if
24.    if gameState[4] = markerOponnent and gameState[8] = markerOponnent then
25.      return new Position(7)
26.    end if
27.    if gameState[6] = markerOponnent and gameState[8] = markerOponnent then
28.      return new Position (9)
29.    end if
       // first sub-strategy against future scissors tactics
30.    if gameState[5] not markerOponnent
         and (     (gameState[1] = markerOponnent
                    and gameState[9] = markerOponnent)
                or  (gameState[3] = markerOponnent
                    and gameState[7] = markerOponnent)
             ) then
31.      set list // holder of positions 2, 4, 6, 8
32.      return randomPositionFromList(list)
33.    end if
       // second sub-strategy against future scissors tactics
34.    if gameState[5] = markerOponnent then
35.      if gameState[1]  = markerOponnent
           or gameState[3]  = markerOponnent
           or gameState[7]  = markerOponnent
           or gameState[9]  = markerOponnent then
36.        set list // holder of positions 1, 3, 7, 9
37.        return randomPositionFromList(list)
38.      end if
39.  end if
     // random choosing of one of the corner cells: 1, 3, 7 or 9
40.  if counter = 1 and not emptyCell then
41.    set list // holder of positions 1, 3, 7 or 9
42.    return randomPositionFromList(list)
43.  end if
     // with last four if statements mirroring corner positions are chosen
44.  if gameState[1] = markerOponnent then
45.    if isTheCellEmpty(new Position(9)) then
46.      return new Position(9)
47.    end if
48.  end if
49.  if gameState[3] = markerOponnent then
       if isTheCellEmpty(new Position(7)) then
50.      return new Position(7)
51.    end if
52.  end if
53.  if gameState[7] = markerOponnent then
54.    if isTheCellEmpty(new Position(3)) then
55.      return new Position(3)
56.    end if
57.  end if
58.  if gameState[9] = markerOponnent then
59.    if isTheCellEmpty(new Position(0)) then
60.      return new Position (0)
61.    end if
62.  end if
end function
```