Test Case Generation by Game Theory

JARON LENDERING, University of Twente, The Netherlands

Model-based testing is a systematic method to create test cases by using a specification model of the system. In earlier papers, the idea of using strategies, for formal games, as test cases was explored. By using a multitude of strategies that have a goal to go to a certain part of the application, the application gets traversed and bugs can be found. This paper compares test generation by different game strategy synthesis techniques to determine which creates better strategies for creating tests. Two synthesizers are compared, Backward Induction (BI) and Simultaneous Move Monte Carlo Tree Search (SM-MCTS). Those strategies are executed on a game, derived from the specification of an executable application created especially for this paper. By using metrics of coverage, speed and test termination factors, I found out that BI works well for small applications and SM-MCTS does not with the used specification. I also predict that SM-MCTS will not work well on any application if it is used as I used it in this paper.

Additional Key Words and Phrases: Model-based testing, 2-Player Concurrent Games, Game theory

1 INTRODUCTION

Testing is an integral part of software development. Without testing, it is impossible to determine the quality of software. However, it also takes a lot of time. That is a problem that model-based testing tries to solve. By specifying the system in a formal model, like an automaton, there are ways to create test cases for the system automatically. A lot of different approaches are being used[4], like Finite State Machines or UML Diagrams, but in this paper, I will create a specification and a game based on that specification. Using the game, I will use game theoretic strategies as test cases for the specification.

A tested system, generally called a System Under Test (SUT), is a system with input and output actions. Input actions are requests to the system and output actions are observable actions of the system. With this in mind, a testing setting can be seen as a 2 player game. The test executes an input action and the SUT executes an output action. Thus, the test can be seen as player 1, the SUT can be seen as player 2 and together they can be modeled as a 2-player game. Multiple authors have already successfully implemented the game theoretic approaches to generate good test cases[3, 8, 11] Also, a generalized model-based testing framework using game-theoretic approaches is outlined in [10]. The writers of [10], show that testing can be modelled as a 2-player concurrent game. However, there has yet to be a study done on how certain strategy synthesis techniques (for 2-player concurrent games) compare to one another. In this paper, I will compare multiple game strategy synthesis techniques on 3 metrics; coverage, strategy creation speed and test termination factors (the reasons why tests finish). In section 9.1, the metrics will be explained further. By creating an executable application myself, I will aim to make an application that is challenging to test with the synthesis techniques, such that a clear view can be given of the strengths and weaknesses of the techniques. The metrics are used as a systematic way to determine which game strategy synthesis techniques are better for testing. This research aims to give a clearer view of what game strategy synthesis techniques would give the best test cases, as determined by the chosen metrics.

1.1 Related work

As mentioned before, multiple authors have already successfully implemented the game theoretic approaches to generate good test cases. In [3], UPPAAL-TIGA [1] is used to test uncontrollable real-time systems. Three novel algorithms for nondeterministic software systems are proposed in [8]. The problem of finding strategies that maximize node coverage, i.e. the number of states visited, is researched in [11].

2 GAMES

As mentioned in the introduction, a generalized model-based testing framework using game-theoretic approaches is outlined in [10]. In [10], many definitions of games and strategies are created. The game, and thus also the play of a game and the winning definition, used in this paper is derived from the one in [10].

The game used is a two-player concurrent game. That means that 2 players are doing an action simultaneously, every turn. A game is formally described by Definition 1.

Definition 1. *A game is a tuple G where,*

- *$G = (Q, q_0, Act_1, Act_2, \Gamma_1, \Gamma_2, Moves)$*
- *Q is a finite set of states*
- *$q_0 \in Q$ is the initial state*
- *Act_i is a finite, non-empty set of actions player i can take*
- *$\Gamma_i : Q \to 2^{Act_i}$ is an enabling condition, which assigns to each state q a set $\Gamma_i(q)$ of actions available to Player i in that state, and*
- *$Moves: Q \times Act_1 \times Act_2 \to 2^Q$ is a function that given the actions of Player 1 and 2 determines the set of next states $Q' \subseteq Q$ the game can be in. We require that $Moves(q, a, x) = \emptyset$ if $a \notin \Gamma_1(q) \lor x \notin \Gamma_2$*

The set of terminal states is defined as $term(Q) = \{q \in Q \mid \Gamma_1(q) = \emptyset \lor \Gamma_2(q) = \emptyset\}$. A state q is terminal if $q \in term(Q)$. Every state has a score $\iota \in \mathbb{R}$, the score of a state is denoted as $score(q)$. Scores are further explained in Section 5

A play is a sequence of states and actions of both players. A play is winning if it visits a winning state in $R \subseteq Q$.

Definition 2. *A play of a game G is an sequence:*
$\pi = q_0 < a_0, x_0 > q_1 < a_1, x_1 > q2...$
with $a_j \in \Gamma_1(q_j)$, $x_j \in \Gamma_2(q_j)$, and $q_{j+1} \in Moves(q_j, a_j, x_j)$ for all $j \in \mathbb{N}$ if $q_j \notin term(Q)$, else $q_(j+1)$ is undefined.
The set of all plays of G is denoted $\Pi(G)$.
We define $\pi_{0:j} \overset{def}{=} q_0 < a_0, x_0 > q_1 < a_1, x_1 > q2...q_j$ as the prefix

TScIT 41, July 5, 2024, Enschede, The Netherlands

of play π up to the j-th state. The set of all prefixes of a set of plays $P \subseteq \Pi(G)$ of G is denoted by $Pref(P) \overset{def}{=} \{\pi_{0:j} \mid \pi \in P,\ j \in N\}$. We define $\Pi^{pref}(G) \overset{def}{=} Pref(\Pi(G))$

Definition 3. *A play $\pi \in \Pi(G)$ of a game G is winning with respect to the goal $R \subseteq Q$, if π reaches some state in R.*

3 MODEL-BASED TESTING

Model-based testing (MBT) is a form of testing that automatically creates test cases using a specification model. The goal of these tests is to find cases where the application does not conform to the specification. If, in the test, the application does an output that does not conform to the specification, the test fails. The definition of conformance will be given in section 3.1. Using MBT techniques, robust tests can be automatically created and manual labour can be reduced.

I use suspension automata (SAs) as system specifications, as seen in [10]. The SAs this paper uses are not identical but are derived from the SA definition in [10]. For a partial function $f : X \rightharpoonup Y$, $f(x)\downarrow$ denotes that $f(x)$ is defined and $f(x)\uparrow$ denotes that f(x) is undefined.

Definition 4. *A suspension automaton (SA) is a 5-tuple $\vartheta = (Q, L_I, L_O, T, q_0)$ where*

- *Q is a non-empty finite set of states,*
- *L_I is a finite set of input labels, and $\mu \in L_I$,*
- *L_O is a finite set of output labels, $\delta \in L_O$, and $L_I \cap L_0 = \varnothing$*
- *T: $Q \times (L_I \cup L_O) \rightharpoonup Q$ is a partial transition function, and*
- *$q_0 \in Q$ is an initial state.*

$\forall q' \in Q,\ T(q', \mu) \rightarrow q'$. I write $L \overset{def}{=} L_I \cup L_O$. For $q \in Q$, the set of enabled inputs and outputs in q are denoted by $in(q) = \{a \in L_I | T(q, a)\downarrow\}$ and $out(q) = \{b \in L_O | T(q, b)\downarrow\}$ respectively.

I assume that any SA has 2 special labels δ and μ to indicate output and input quiescence respectively. I define quiescence as having no observable action $x \in L$, input quiescence as having no observable action $y \in L_I$ and output quiescence as having no observable action $z \in L_O$. A SUT can have a non-observable output action, sometimes the SUT does not respond after an input action is done by the tester. An input action can also be non-observable. Many output actions can be done in a row, but no input action has to be done in between. For example, when downloading an application, a lot might be happening on screen while the user is not doing an input. If a user of the application does not do anything, so they do a quiescent action, they do not express the desire to change state. Following that assumption, I assume that when the input is quiescent, it cannot initiate a state change. If a state change does occur while the input is quiescent, it is always initiated by the output. Also, the user can at any time choose to just not do an action, so that is why $\forall q' \in Q,\ T(q', \mu) \rightarrow q'$.

3.1 Conformance relation

Test cases will test if the SUT conforms to the specification. The conformance relation used in this paper is called IOCO [9]. IOCO is short for input-output conformance. IOCO allows for more inputs and fewer outputs than the specification. This practically means that all input actions are permitted, and if the input action is not in the specification, every output action in return would be allowed. However, as long as the input actions are part of the specification, any output action must also be in the specification. This allows the specification designer to design a specification that entails only a part of the SUT, and still create test cases for that part.

3.2 Input-Output conflicts

All states enable Input and Output actions, but only 1 action can be done. A conflict arises when the tester wants to do an input action and the SUT an output action. Multiple assumptions can be used to solve such a conflict. I have chosen to use an output-eager approach, an approach where the output action will always be executed if the output action is not quiescent. The reasoning behind the chosen approach will be described in section 7.2

3.3 Specifications to games

In [10] a translation from specifications to games is proposed. The writers created a definition for the translation. I create a definition derived from theirs, working with my definition for a specification.

Definition 5. *Let $\vartheta = (Q, L_I, L_O, T, q_0)$ be an SA. The game underlying ϑ is defined by $G_\vartheta = (Q, q_0, Act_1, Act_2, \Gamma_1, \Gamma_2, Moves)$ where:*

- *$Act_1 = L_I$ and $Act_2 = L_O$*
- *for all $q \in Q$, we take $\Gamma_1(q) = in(q)$ and $\Gamma_2(q) = out(g)$*
- *$Moves\ :\ Q \times Act_1 \times Act_2 \rightarrow 2^Q$ encodes a different test assumption to handle input-output conflicts.*

The specification (Definition 4) only accepts an input or an output label, but the game used is a concurrent game, so an input and output action is done at the same time. So using a chosen test assumption, only 1 action is executed on the specification. I denote the combination of an input and output action, from which 1 action will be chosen following the test assumption, a transition.

4 STRATEGIES

Strategies are functions that decide what action a player does at any turn in the game. A strategy is called winning, if a player always wins if they use that strategy, no matter what the other player does. The definition for strategies and the translations between strategies and test cases are derived from the definition given in [10]

Definition 6. *A strategy for player i in game G is a partial function σ_i: $\Pi^{pref}(G) \rightarrow Act_i$, such that σ_i is either undefined or $\sigma_i(\pi) \in \Gamma_i(q')$ for any $\pi \in \Pi^{pref}(G)$, where q' is the last reached state of π.*

In Definition 5, specifications are translated to games. Strategies can also be seen as test cases on those specifications. First I define what a test case is. A test case is a sequence of actions on which mutually exclusive failing and passing conditions are applied. This means that certain sequences of actions will result in a fail and certain sequences of actions will result in a pass. For example, a passing condition could be a sequence of actions that reaches the winning state and a failing condition could be a sequence of actions that reaches a non-winning terminal state. A test case has at least 1 passing and 1 failing condition and a test always ends in a pass or a fail.

I will translate a strategy to a test. A strategy returns at every state a valid input action $x \in L_I$ or undefined. If it returns undefined, the test fails because the winning state has not been reached and the strategy does not know what next action to take. If it returns an action, it can be used in the SUT to see what output action the SUT does. If the SUT returns an output that does not correspond with the specification, the test fails. When a winning state is reached, the test passes. This defines a strategy as a test case whose goal is to go to a certain state in the application, and which tests if the SUT corresponds to the specification.

4.1 Strategy creation

To create multiple strategies, strategy synthesizers are used. Strategy synthesizers are algorithms that can create different strategies, depending on what goal is defined. In this paper, the only defined goals are the winning states $R \subseteq Q$. For every winning state, a new strategy is created by the strategy synthesizer. When creating strategies, we run the strategy synthesizer as many times as preferred. In every run a different state can be the winning state, to test the reachability of different states. This way several different test cases are created to, in the end, try to test enough different input and output actions such that any potential mistakes in the SUT can be found. In this paper, I try to compare different synthesizers. I will run different synthesizers with many different states as winning states and compare the strategies created by the synthesizers.

5 SCORES

Every state of the game gets a score. Those scores are being used to indicate how good entering a certain state is. Strategy synthesizers use those scores to decide what actions are the best to take in specific states, thus creating good strategies.
The scores are defined by the creator of the game. I have chosen a very simple scoring system. Every state gets a score of 0, except for the winning state. The winning state gets a score of 1. This is the easiest way of giving scores because the only information needed is which state is the winning state. However, strategy synthesizers have a harder time creating good strategies, because they have less information to work with. As long as the winning state is not evaluated by the synthesizer, all actions will be considered equally beneficial, because every evaluated state will have a score of 0. No state will be considered to be better than another one.
Nevertheless, I have chosen this simple scoring system. No hard calculations have to be done to set these scores and all the computing time can be given to the strategy synthesizers. Also, because it is simple, it is not made specifically for a certain synthesis technique. This gives the chosen techniques a fair chance.

5.1 Future work

In future work other scoring techniques can be tested, to see if they can be viable to get better strategies, for testing, from certain synthesising techniques. One scoring technique that could be tested, is giving higher scores depending on the proximity to the winning state. This might give fast, but less thorough, synthesizers a better chance of getting a good path to the winning state, because information about the positioning of the winning state is also available in non-winning states.

6 STRATEGY SYNTHESIZERS

For this paper, I have chosen 2 strategy synthesizers. Backward induction (BI) and Simultaneous Move Monte Carlo Tree Search (SM-MCTS).

6.1 Backward Induction

6.1.1 Original Algorithm. Backward Induction is one of the first algorithms for solving 2 player turn-based games. It works by trying 1 specific set of moves until a terminal state is reached. When it reaches a terminal state, it goes 1 state back and chooses a different action until it reaches a terminal state again. This will be done until all states are traversed. This is called dept-first search. When a terminal state $q_t \in Q$ is reached, the score is propagated to one state before, $q_{t-1} \in Q$. Because at least 1 new state can be reached from any non-terminal state, at least one score will be propagated to q_{t-1}. The highest score that is propagated back to q_{t-1} becomes the new score for q_{t-1} and that score will be propagated to q_{t-2}. This will be done until the scores are propagated to the initial state. As mentioned earlier, originally BI was used for turn-based games. Nevertheless, by treating the combined move of both players as 1 move, it can be used for concurrent games[2]. The algorithm used in this paper is derived from [2] with some minor changes to be able to incorporate loops in the specification. The new algorithm can handle loops, because when a state is reached, BI will check if it has already seen that state. If BI has seen the state before, it takes the current score of the state and will not iterate further.

Algorithm 1 bi_strategy_synthesize

```
1:  input: q - current state
2:  if $q \in R$ then
3:      return score(q)
4:  end if
5:  if $q \in Q^{seen}$ then
6:      return $score^{high}(q)$
7:  end if
8:  $A_I^{high} \leftarrow None$
9:  $score^{high}(q) \leftarrow 0$
10: for $A_I \in in(q)$ do
11:     $score^{A_I} \leftarrow 0$
12:     for $A_O \in out(q)$ do
13:         $q' \leftarrow Moves(q, A_I, A_O)$
14:         $score^{A_I} \leftarrow score^{A_I} + bi\_strategy\_synthesize(q')$
15:     end for
16:     if length(out(q)) > 0 then
17:         $score^{A_I} \leftarrow score^{A_I} / length(out(q))$
18:     end if
19:     if $score^{high}(q) < score^{A_I}$ then
20:         $score^{high}(q) \leftarrow score^{A_I}$
21:         $A_I^{high} \leftarrow A_I$
22:     end if
23: end for
24: $\sigma(q) = A_I^{high}$
25: return $score^{high}(q)$
```

6.1.2 New Algorithm.

6.1.2 New Algorithm. $R \subseteq Q$ is the set of all winning states. Q^{seen} is the set of traversed states. σ is the strategy that is being created. A_I^{high} stores the action that can get the highest possible score. $score^{high}(q)$ stores the highest score gotten. If the current state is already seen earlier, $score^{high}(q)$ is set in an earlier iteration of the algorithm. All these definitions are also used by any other algorithm in this paper.

Algorithm 1 describes a BI algorithm where an input action is first considered, and every output action possible with the chosen input action is considered. An input and output action together initiate a state change. For any input action, on any state, the total score is divided by the number of output actions because I assume nothing about the strategy of the SUT. Thus the chance that the SUT chooses a certain output action is equal to any other output action. For every state, the best input action will be saved. Scores are only saved for input actions because input actions are the only actions done by the tester. When a terminal state is reached, $out(q) = \varnothing$, so the algorithm will go in the second for loop and $bi\_strategy\_synthesize(q')$ is not called again.

The algorithm tries more actions until a terminal or winning state is reached, instead of only when a terminal state is reached. This also means that not all states have to be traversed. For example, if the initial state is the winning state, it will immediately stop and no other states must be traversed. For the remainder of the paper, any reference to BI will indicate algorithm 1.

6.2 Simultaneous Move Monte Carlo Tree Search

6.2.1 Original Algorithm. Monte Carlo tree search (MCTS) is an often used algorithm for big state spaces, a copious amount of different states reachable within the game, because it does not need to go through the whole state space to return a result. A version of MCTS has been successfully used to create very strong Machine-learning algorithms for the game Go[5]. Like BI, MCTS was also developed for turn-based games. However, a version created for concurrent games is used in [7], called Simultaneous Move Monte Carlo Tree Search (SM-MCTS).

SM-MCTS works by first choosing an initial state from the current game. Starting with the initial state, it will know if all the transitions from that state are explored at least once. That means that with every transition, a path until a terminal node is explored. If not all transitions are explored, the algorithm will choose a random path until a terminal state, using a ROLLOUT function, and return the score acquired in that path. If all transitions are explored, it will choose one of the transitions to transition the game to the next state, using a SELECT function. Using that new state, the algorithm will again see if a transition from that state has been unexplored. This will continue until a state is reached which is either terminal or has unexplored transitions. The scores that are retrieved by choosing certain actions are saved. The SELECT function will then use those scores to choose a transition to further explore. By doing this algorithm over and over again, more and more states are explored. The strength of this algorithm is that the Tester can choose the amount of iterations the algorithm needs to run. More iterations mean more exploring, but it also uses more resources. The goal of SM-MCTS is to find a list of actions that gives the highest score possible, and thus the best strategy. The algorithm used in this paper is derived from the basic one, from [7], but with added features.

Algorithm 2 SM_MCTS_strategy_synthesize

```
1: input: q - current state
2: if $q \in term(Q)$ or $q \in R$ then
3:     return $score(q)$
4: end if
5: if $A_I^{unseen} \neq \varnothing$ then
6:     $A_I, A_O \in A_I^{unseen}$
7:     $q' \leftarrow Moves(q, A_I, A_O)$
8:     $rollout\_score \leftarrow$ ROLLOUT($q'$)
9:     UPDATE(q,$A_I$,$A_O$,$rollout\_score$)
10:    return $rollout\_score$
11: end if
12: $A_I, A_O \leftarrow SELECT(q)$
13: $q' \leftarrow Moves(q, A_I, A_O)$
14: if $recursion\_left > 0$ then
15:     $select\_score \leftarrow$ SM_MCTS_strategy_synthesize(q', recursion_left - 1)
16: else
17:     $select\_score \leftarrow$ score(q')
18: end if
19: UPDATE(q,$A_I$,$A_O$,$select/_score$)
20: return score
```

6.2.2 New Algorithm. A_i^{unseen} is the set of all actions of player i that are not yet explored.

Algorithm 2 works the same as the original algorithm from [2], but with a few differences. The original ROLLOUT function chooses a random path, but some state spaces are vast so it will not reach a terminal state. So, in my algorithm, it can also converge when the maximum number of iterations has been reached. This way it will not go on until it eventually crashes because of too much memory usage, and it will not get stuck in loops. Also, it will create a faster algorithm, if the max iteration count is set low. The second difference is that the algorithm also converges at winning states because SM-MCTS doesn't have to search further if the winning state has been found, even if the winning state is not terminal. I also set the maximum recursion dept for the algorithm, which practically means how many states the algorithm may explore from the initial state (maintained by recursion_left), for the same reason. The last difference is how new states are added to the tree. In my version of SM-MCTS, states are only added if they have not been reached earlier. This makes the algorithm more efficient because identical parts will not be traversed multiple times and it makes sure that the algorithm will not be stuck in a loop.

The SELECT algorithm used is a decoupled upper-confidence bound applied to trees (DUCT). DUCT is the most common selection function for SM-MCTS [2]. For each player, per action, the number of times that transition is done is saved. Also, every time an action is done, the obtained score will be added to the total score for that action. This is done by the UPDATE function. When a new action has to be selected by the SELECT function, both actions (for the

Tester and the SUT) will be evaluated separately. The actions with the best average score (score divided by the number of times the action is done) will be selected. If multiple actions have the highest score, a random action will be selected.
The aforementioned algorithm is exactly how my SM-MCTS algorithm is supposed to work. However, while testing I found one problem. The quiescent input action of the initial state seems to sometimes get a score above 0, while no action at any other state gets a score above 0. I could not find out why, so I chose to give the quiescent action of the initial state always a score of 0. Executing the quiescent action of the initial state brings the specification to the initial state, so executing that quiescent action is never needed. For the remainder of the paper, any reference to SM-MCTS will indicate algorithm 2.

6.2.3 Future work. DUCT is an often used selection algorithm but is probably not the best one for creating strategies as test cases with reachability as the goal. Because only 1 state is the winning state, most states get a score of 0. That means that there is a big chance that all actions will get a score of 0 in the first rollout. If that is the case, with DUCT a random action is chosen. This might result in some actions getting chosen far more often than others. If unlucky, this could lead to a strategy where the best action is never explored. To counteract this, the select function could be changed. I propose to implement a variant to regret-matching (RM)[6].
RM can be used to have more control over the chosen actions, by changing the probability an action is chosen. The algorithm I propose was successfully created and explained thoroughly in[2]. The idea is that for every possible combination of input and output actions, the number of visitations and the cumulative score are saved. Also, for every action, a regret value is saved. It gets increased when an action is not chosen. This increases the probability of actions that are chosen less than others, so there is a lower chance that an action is not explored at all. Nevertheless, the probability of actions that got a high score on earlier iterations will still be high. The chance that the best strategy uses an action that already got a high score is higher than the chance that it uses an action that got a low score after all.

7 APPLICATION AND SPECIFICATION

7.1 Specification

To test the strategies, I made a specification with an executable application that adheres to it. It consists of 3 parts. Firstly, a loop that loops between 3 states, is further called loop path. Secondly, a path that has a lot of different choices. Every choice the tester makes brings the system to a different state, and from that different state, new choices can be made, until a terminal state is reached. This part of the specification is further called choice path. Lastly, there are multiple paths that all go to the same end-state, further called line path. By adding these 3 parts to my application, I evaluate important parts that could be hard for strategy synthesizers, but important in real-world applications. The total number of states in the specification is 846. The specification is shown in Figure 1.

Loop path highlights having an infinite loop in the specification. Stratgies that only converge on a terminal state will never converge, because it will loop. It also has 2 states, middle_loop and side_loop, that both have actions to loop1 and loop2. However, in middle_loop, the SUT is in full control and decides if or when the game goes to loop2. In side_loop the tester is in full control. Also, side_loop has a quiescent output action that goes to middle_loop. So if the tester and the SUT do a quiescent action in side_loop, the next state will be middle_loop. Choice path highlights having a big application. Big, complicated applications have a lot of states with a lot of choices that can be made. This could be complicated for some strategy synthesizers because the winning state could be very hard to find. Line path highlights the multitude of ways that an identical result could be reached using different paths. This makes it easy for a strategy synthesizer to find a path to a winning state, but pretty hard to evaluate all the paths.

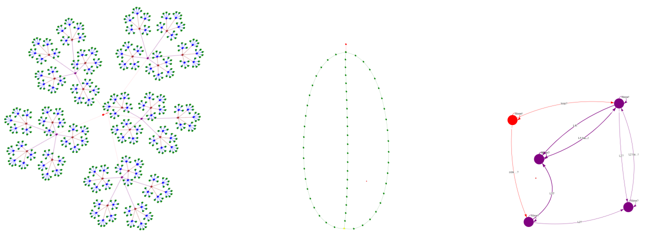
Fig. 1. The left image is the specification of the choice part. Every colour represents a new dept of the choice tree. The middle image shows the specification of the line path and the right image shows the specification of the loop path. The real specification consists of all three parts, all connected to one initial state.

7.2 Application

To make the test environment as realistic as possible, I made an executable application to execute the test cases on. That application runs in parallel with the strategies executed on the game, defined by a specification. I denote this as a realistic test environment because only a few requirements are defined for an application to run with my test environment. An application has to,

- have a specification, as described in Definition 4,
- have 1 explicit input and 1 explicit output action function, such that the input and output can be read by the testing framework,
- have any action that the application does be an explicit output, so also when the application does nothing (a quiescent action). This is important because the test framework has to know exactly what is happening in the application.

To use strategies, the testing framework executes those strategies on a game representation of the specification of the application. The game used is a 2-player game, however, my application does only 1 action at a time. Because I use single explicit input and output functions, I assume that only 1 input and 1 output can be done at a time. So if a parallel application is evaluated, internally all actions will be done sequentially. Because only 1 action can be done at a time, the input or the output action has to be chosen. I have opted for the output-eager approach. This means that if the output is not a quiescent action, the output action will always be chosen. Before I explain my reasoning, I first need to explain how the application works. When the application starts, it expects an input action and it will execute an output action. If an input action is done that is not in the specification, the program will exit. I do that, such that non-specified actions could never go to a state in the specification. It wouldn't change anything while testing, but I did it for clarity. Every time input is expected, output has to be expected too, and vice-versa. However, sometimes multiple input or output actions might have to be done in a row. If there is no non-quiescent output action, the text *None* will be returned by the application. I do this because without a resulting output that is readable, I can never certainly know if no output action will ever be done. The chance that the application is just slow is always a possibility. If an input action is quiescent, it will also relay *None*. This makes keeping track of in what state the application is easier because the actions are always explicit, so state changes are also explicit. It also gives me full control over the execution of the application, since the application can't proceed until an input action is done. This brings me to why I chose an output-eager approach. Inputs are chosen by the strategy after an output action is already done. Had I chosen a variant that prefers an input action over an output action, the output action had to be reverted after it had been done. That makes creating an application harder and less realistic because a reverting function had to be added at every point. In addition to that, the output shown by the application would not be consistent with the output evaluated. In the current version of the application, the output is first displayed, and afterwards, it is evaluated. So if an output action would be reverted after it is displayed, an output is displayed but, from the perspective of the application, it is not done.

8 TESTING FRAMEWORK

The testing framework uses the created strategies and the specification to test the application. It will first check if the current state is winning or terminal. Subsequently, it will read the last output action returned by the application and an input action from the currently running strategy is obtained. The specification is advanced by the input or the output action, chosen in an output-eager manner. If the action is not possible in the specification, it will raise an error. After checking if the action is in the specification, the input action is executed on the application. This sequence of actions will be done until an output is done that does not conform with the specification, a winning or terminal state is reached, the strategy can't generate an input action for the current state or the same state is traversed too often (In this paper, more than 5 times is used). The sequence of actions mentioned above is executed for every strategy a chosen strategy synthesizer creates. After all the strategies are executed, metrics about coverage and strategy creation speed are returned.

9 METHODOLOGY

9.1 Metrics

Using the aforementioned test framework, application, specification and strategy synthesizers, I did a multitude of tests to determine the qualities of both strategy synthesizers. To measure the qualities, I have defined 2 metrics. Strategy creation speed and coverage and termination factors.

9.1.1 Strategy Creation Speed. The time efficiency of testing is extremely important because the longer the test time is, the longer programmers will be unsure if their application works how it is supposed to. There are two main parts of time efficiency while testing using game strategies. The strategy creation time and the testing time. Strategy creation time is the time it takes to create all the strategies, i.e. test cases, that the synthesizer had to create. In this paper, I only look at strategy creation time, because testing time was too sporadic.

9.1.2 Coverage. In this paper, coverage is defined as the number of states that are traversed while creating a strategy. Coverage is a metric in the same realm as strategy creation speed because it is also about efficiency. All strategies have the same goal, going to the winning state. Every traversed state that fails to advance the tester to the winning state, is a waste of resources. So by keeping the coverage low, but still reaching the winning state, the strategy becomes more efficient.

9.1.3 Termination Factors. The last metric I will cover is the factors why a strategy won or lost, i.e. terminated. Some factors are outside the control of the strategy. For example, one part of the loop can only be accessed if the SUT does a specific output action. As long as that doesn't happen, the tester can never reach that state. However, the majority of the time the strategy is in control. The most important termination factors are Terminal State (TS), Duplicate States (DS) and winning. TS happens when the game reaches a terminal state which is not the winning state, which can happen if the strategy does the wrong input action. DS happens when the strategy is stuck in a loop, so the game reaches the same state more than 5 times. Winning is the most important factor because it means that the winning state has been reached.

9.2 Process

To test the quality of the strategy synthesizers, every strategy created by a strategy synthesizer is run once using the specified application, described in Section 7. Because of the large amounts of states the specification has, a lot of states are similar. Thus, a lot of created strategies are also similar. So even though the created strategies are only ran once, because of the similarities in the created strategies, the results are still significant.
While testing, I ran a lot of different synthesizers, or synthesizers with different constants (see Section 10) for more information), in parallel. Because of that, the CPU usage was 100%. This made the total execution time of all strategies really sporadic, but because creating the strategies did not take up too many resources, it was less sporadic the results were still significant.

10 RESULTS

10.1 Backward Induction

The first strategy examined is Backward Induction (BI). Backward induction searches thoroughly, never stopping until it reaches the winning state or it has traversed every state in the game. That makes the algorithm extremely reliable.
In testing, it turned out that backward induction always wins, if it can win. When testing only parts of the specification (line part, choice part and loop part), or when testing the whole specification, it always won. Even when there are 2 winning paths, as in the loop

section, it still chooses the path that gave the tester a 100% chance of winning.

All the created strategies combined took 12 seconds to generate, with BI. It traversed between 800 and 846 states for every strategy where the goal state was further than 1 action away from the initial state. This also shows that the biggest strength of BI, the thoroughness, is also its biggest weakness. BI traversed, for most strategies, almost all of the 846 states. The fluctuation in the number of traversed states comes from the exact position of the winning states. If the winning state is in a position that gets traversed earlier, fewer states are traversed in total.

10.2 Simultaneous Move Monte Carlo Tree Search

SM-MCTS is more complicated than BI because it gives the possibility to set the max rollout iterations ($Roll_{it}$), the most recursion dept (RD) and the max iteration count of the algorithm (alg_{it}). I tested SM-MCTS with a multitude of combinations. For the remainder of the paper, whenever the maxima are given, it will be in the format ($Roll_{it}$, RD, alg_{it}) unless specified differently.

The full specification will be evaluated. The number of strategies created is 846 because there are 846 states in the specification.

10.2.1 Termination factors And Creation Time. As shown in Table 2, the SM-MCTS algorithm does not work extremely well. The worst version is when all of the maxima were low (12,50,30). The choice part did the worst. This most likely happened because there are fewer choices to be made in loops and lines than in choices, so the chance that the winning state is quickly found is smaller in choices. When only the alg_{it} was increased, the algorithm became better, however, the other strategies still terminated because of continuously reaching the same state. It also took almost 104 seconds to create the strategies. No synthesizer created strategies of which more than 25% won, but there is a trend in the amount of winning strategies. The higher alg_{it}, the more winning strategies. That most likely happens because more explored transitions mean a bigger chance that a winning path is found.

In Table 2, it is shown that strategies with winning states in the choice part of the specification mostly lose. That is expected because there are a lot of different choices being made in that part, so it is hard for SM-MCTS to find the winning state. And when it doesn't even find the state, it will certainly lose. However, strategies can also lose in the lines and loop part. That is peculiar because the winning state is pretty easy to find in those parts, not many choices have to be made.

Looking at the created strategies, it can be seen that the quiescent input action gets the same score as a non-quiescent input action. That means that it will be totally random if a strategy does a quiescent or non-quiescent input action. The scores are the same because ROLLOUT returns the same score no matter if a quiescent input action is done or not. For example, ROLLOUT returns the score 1 for path π that starts with input action α. If a quiescent input action μ is done, the current state does not change. So after μ, π can still be executed. So if μ is done before π, μ receives a score of 1. Without μ, α gets a score of 1. So the strategy considers both actions as of equal quality. However, if μ is done 6 times, the same state is traversed 6 times, and the strategy will lose. To counteract that I made a second SM-MCTS strategy, called $SM - MCTS_{\not\mu}$. This algorithm is identical to SM-MCTS, however, it will always prefer a non-quiescent action above a quiescent action. $SM - MCTS_{\not\mu}$ will be further explored in Section 10.3.

10.2.2 Coverage. Table 1 shows a clear pattern in average coverage (the average coverage of all strategies created using certain values for the constants). When alg_{it} increased, the average coverage also does. After inspecting the SM-MCTS algorithm, it is clear why. One iteration of the SM-MCTS algorithm is done after a winning state is found, a terminal state is found or a state with an unexplored transition is found. When a terminal or winning state is found, at most 1 new state is explored in that iteration, the winning or terminal state (if the winning or terminal state is found in an earlier iteration, no new states are explored). When a state with an unexplored transition is found, 1 new state will be explored. The transitions explored in the rollout are not saved, so they are not counted in the coverage of the final strategy. This means that at most 1 state is added every iteration, so the coverage scales with alg_{it}.

$Roll_{it}$	RD	alg_{it}	Results		
			Win Strategies	Time (s)	Av. Coverage
12	50	30	38	5	20
300	150	30	45	20	20
300	150	100	107	82	45
300	150	200	156	170	75
12	50	300	203	104	110

Table 1. Table of the amount of winning strategies, strategy creation times and the average coverage for all created strategies for different combinations of $Roll_{it}$, RD, and alg_{it}

$Roll_{it}$	RD	alg_{it}	Results		
			Line part	Loop part	Choice part
12	50	30	32%	75%	2%
300	150	30	45%	75%	2%
300	150	100	74%	100%	7%
300	150	200	67%	50%	14%
12	50	300	67%	100%	19%

Table 2. Table of the win percentage per specification part for SM-MCTS, for different combinations of $Roll_{it}$, RD, and alg_{it}

10.3 $SM - MCTS_{\not\mu}$

$SM - MCTS_{\not\mu}$ is SM-MCTS but with a preference for non-quiescent actions. The strategy generation of $SM - MCTS_{\not\mu}$ does not change relative to SM-MCTS, since the preference is implemented after the strategy creation.

When the testing framework expects an input action from the strategy, the strategy will choose one of the actions with the highest score that are non-quiescent, if there are non-quiescent actions. If the only input action available is quiescent, it will choose the quiescent action. However, the number of winning strategies has increased. As seen in Table 3, win rates in the loop and line part

have increased significantly. The loop part increased because the tester kept control by doing non-quiescent actions. The SUT can't bring the game to a state where it gets all the control.
The win rate in the line part increased because the multiple duplicate states termination is only reached when the synthesizer doesn't get enough resources to create a strategy that goes to the winning state. So even though strategies created with a synthesizer with only high $Roll_{it}$ and RD or alg_{it} still lose often, the strategies created when the synthesizer gets a high $Roll_{it}$, RD and alg_{it} are really good for winning states in the line part.
The win rate in the choice part has not increased because, often, no winning path is found. If alg_{it} would be higher, more states would be explored, so the win rate would increase.

$Roll_{it}$	RD	alg_{it}	Results		
			Line part	Loop part	Choice part
12	50	30	32%	75%	2%
300	150	30	50%	100%	1%
300	150	100	98%	100%	8%
300	150	200	100%	100%	14%
12	50	300	72%	100%	19%

Table 3. Table of the win percentage per specification part for $SM-MCTS_{\not\mu}$, for different combinations of $Roll_{it}$, RD, and alg_{it}

10.4 Comparison

My experiments clearly showed the power of BI, and it easily trumped SM-MCTS. BI was faster than SM-MCTS and created strategies that always won. Nevertheless, SM-MCTS also had some qualities. The most prominent one was that the coverage was less. This shows that, if optimized, SM-MCTS could be faster, because it visits fewer states. In my application, it still visits a lot of states double, but with optimizations that doesn't have to be the case. Also, SM-MCTS focuses most of their resources in one direction. If the path to the winning state starts with action α, it thoroughly checks if any other paths starting with α also win. This makes the chance that SM-MCTS checks states that can never reach the winning state small. Still, the strategies created by MCTS lose often, mostly when a lot of choices are involved. MCTS has been proven to work well in games where every state can get a positive score, like in Go. Every position can be given a score, so MCTS is not blind until it finds a winning state. In my tests, SM-MCTS is blind until it reaches the winning state. Because of that, I think that MCTS will never be a very good algorithm for creating strategies as test cases, as long as the scoring is defined how I defined it in this paper. An improvement could be, to give scores depending on how many actions a state is removed from the winning state.
In conclusion, the way I gave scores and created test cases gives far better results for BI. If the SUT is small, I would recommend BI. SM-MCTS might get better results if it is more optimized, but my basic implementation performs too poorly for real-world use. To use strategy synthesize techniques in big applications, with large specifications, other strategies have to be implemented. However, I don't think that there is a synthesizer that is fast and creates winning strategies for a specification with only a positive score on the winning state. Other score systems, or other ways of using the strategies, have to be created to be able to scale the specification up. One way the test performance might be increased without changing the scoring system is to start 2 strategies. One at the initial state, going to the winning state, and one going in reverse from the winning state and let them meet in the middle.

11 CONCLUSION

In this paper, I explored which strategy synthesizing techniques are the best for testing software applications by doing a case study on an executable application. This was done following the [10] about translating strategies into test cases. I started by doing a literary review on different strategy synthesising techniques and eventually found 2 promising techniques; Backward Induction (BI) [2] and Simultaneous Move Monte Carlo Tree Search (SM-MCTS) [7].
After the literary review, I created a testing framework that could use strategies to test an executable application. The application was simple, with 3 main parts. One for making choices, one for having a loop and one for having a lot of states in a row. Lastly, I also created 2 strategy synthesizers, BI and SM-MCTS, that are derived from the BI and SM-MCTS algorithms explained in [2] and [7] respectively. I changed the algorithms a little bit to create a bit better algorithms for my score system in the case of BI. In the case of SM-MCTS, I changed the algorithm from [7] to prevent crashes and give myself more control. The goal of the created strategies was to try and reach a certain state deemed winning. The synthesizers, applications and specifications are created in a way that allows for new synthesizers, applications and specifications to be created. That way it could be used, with maybe some changes, for further case studies.
After the testing framework was fully built, the synthesizers could be evaluated. My experiments showed that BI was the superior synthesizer for a simple, small application. The application was relatively small, with around 846 states and for most states, only 1 sequence of actions could reach the state. BI was fast, creating all strategies in 12 seconds, and it always reached the winning state. SM-MCTS did not work well, it was slower and it failed more often. I improved the algorithm by making it prefer non-quiescent actions over quiescent actions, but it did not make it much better. In the end, I think that BI and SM-MCTS would never work in big real-world applications, the way I have implemented them. To use the game strategies as tests in real-world applications, more optimized ways of giving scores or different ways to implement the strategies as tests should be explored.

11.1 Future Work

I have already touched on future work multiple times in this paper because I think that there is a lot more research needed to determine the functionality of certain game strategy synthesis techniques. SM-MCTS can be further improved to work better as a test case, and different ways of distributing scores can be researched. Also, different synthesis techniques can be tested or whole different ways of implementing the strategies. In this paper, I focused on reachability goals and started every test from an initial starting state. However, different goals or different starting states could be used. Running multiple strategies in unison and combining the strategies in 1 test case is also a possibility that could not be explored in this paper.

12 AI DISCLOSURE

During the preparation of this work the author used Grammarly in order to check and fix the grammar and to change some sentences to make them more clear. After using this tool/service the author reviewed and edited the content as needed and takes full responsibility for the content of the work.
During the preparation of this work the author used chat-gpt in order to find synonyms for words. After using this tool/service the author reviewed and edited the content as needed and takes full responsibility for the content of the work.
During the preparation of this work the author used chat-gpt in order to learn about certain structures in python (like Threads) and to find the cause and sometimes fix for bugs. After using this tool/service the author reviewed and edited the content as needed and takes full responsibility for the content of the work.
During the preparation of this work the author used chat-gpt in order to create tables and figure layouts in latex to use in the paper. After using this tool/service the author reviewed and edited the content as needed and takes full responsibility for the content of the work.
During the preparation of this work the author used chat-gpt in order to learn about certain packages and keywords for latex. After using this tool/service the author reviewed and edited the content as needed and takes full responsibility for the content of the work.

REFERENCES

[1] Gerd Behrmann, Agnès Cougnard, Alexandre David, Emmanuel Fleury, Kim G. Larsen, and Didier Lime. 2007. UPPAAL-Tiga: Time for Playing Games!: (Tool Paper). In *Computer Aided Verification*, David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi, Gerhard Weikum, Werner Damm, and Holger Hermanns (Eds.). Vol. 4590. Springer Berlin Heidelberg, Berlin, Heidelberg, 121–125. https://doi.org/10.1007/978-3-540-73368-3_14 Series Title: Lecture Notes in Computer Science.

[2] Branislav Bošanský, Viliam Lisý, Marc Lanctot, Jiří Čermák, and Mark H. M. Winands. 2016. Algorithms for computing strategies in two-player simultaneous move games. *Artificial Intelligence* 237 (Aug. 2016), 1–40. https://doi.org/10.1016/j.artint.2016.03.005

[3] Alexandre David, Kim G. Larsen, Shuhao Li, and Brian Nielsen. 2008. A game-theoretic approach to real-time system testing. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE '08)*. Association for Computing Machinery, New York, NY, USA, 486–491. https://doi.org/10.1145/1403375.1403491 event-place: Munich, Germany.

[4] Arilo C. Dias Neto, Rajesh Subramanyan, Marlon Vieira, and Guilherme H. Travassos. 2007. A survey on model-based testing approaches: a systematic review. In *Proceedings of the 1st ACM international workshop on Empirical assessment of software engineering languages and technologies: held in conjunction with the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE) 2007*. ACM, Atlanta Georgia, 31–36. https://doi.org/10.1145/1353673.1353681

[5] Sylvain Gelly, Levente Kocsis, Marc Schoenauer, Michèle Sebag, David Silver, Csaba Szepesvári, and Olivier Teytaud. 2012. The Grand Challenge of Computer Go: Monte Carlo Tree Search and Extensions. *Commun. ACM* 55 (March 2012), 106–113. https://doi.org/10.1145/2093548.2093574

[6] Sergiu Hart and Andreu Mas-Colell. 2000. A Simple Adaptive Procedure Leading to Correlated Equilibrium. *Econometrica* 68, 5 (Sept. 2000), 1127–1150. http://www.jstor.org/stable/2999445 Publisher: [Wiley, Econometric Society].

[7] Marc Lanctot, Viliam Lisý, and Mark H. M. Winands. 2014. Monte Carlo Tree Search in Simultaneous Move Games with Applications to Goofspiel. In *Computer Games*, Tristan Cazenave, Mark H.M. Winands, and Hiroyuki Iida (Eds.). Springer International Publishing, Cham, 28–43. https://doi.org/10.1007/978-3-319-05428-5_3

[8] Lev Nachmanson, Margus Veanes, Wolfram Schulte, Nikolai Tillmann, and Wolfgang Grieskamp. 2004. Optimal strategies for testing nondeterministic systems. *ACM SIGSOFT Software Engineering Notes* 29, 4 (July 2004), 55–64. https://doi.org/10.1145/1013886.1007520 Publisher: Association for Computing Machinery (ACM).

[9] Jan Tretmans. 2008. Model Based Testing with Labelled Transition Systems. In *Formal Methods and Testing: An Outcome of the FORTEST Network, Revised Selected Papers*, Robert M. Hierons, Jonathan P. Bowen, and Mark Harman (Eds.). Springer, Berlin, Heidelberg, 1–38. https://doi.org/10.1007/978-3-540-78917-8_1

[10] Petra van den Bos and Marielle Stoelinga. 2018. Tester versus Bug: A Generic Framework for Model-Based Testing via Games. *Electronic Proceedings in Theoretical Computer Science* 277 (Sept. 2018), 118–132. https://doi.org/10.4204/eptcs.277.9 Publisher: Open Publishing Association.

[11] Farn Wang, Sven Schewe, and Jung-Hsuan Wu. 2015. Complexity of node coverage games. *Theoretical Computer Science* 576 (April 2015), 45–60. https://doi.org/10.1016/j.tcs.2015.02.002 Publisher: Elsevier BV.