Tester versus Bug:
A Generic Framework for Model-Based Testing via Games

Petra van den Bos [1] [*]   Marielle Stoelinga [1] [2] [*] [†]

petra@cs.ru.nl   marielle@cs.utwente.nl

[1]Institute for Computing and Information Sciences, Radboud University, the Netherlands

[2]Formal Methods & Tools, University of Twente, the Netherlands

Abstract. We propose a generic game-based approach for test case generation. We set up a game between the tester and the System Under Test, in such a way that test cases correspond to game strategies, and the conformance relation ioco corresponds to alternating refinement. We show that different test assumptions from the literature can be easily incorporated, by slightly varying the moves in the games and their outcomes. In this way, our framework allows a wide plethora of game-theoretic techniques to be deployed for model based testing.

Testing is a widely practiced method to validate the correctness of a system. By exercising a System Under Test (SUT) to a large number of test cases, the tester gets insight in the quality of the system, and especially whether or not a system conforms to its specification. At the same time, testing is very expensive, often requiring 30% of the development cost of a system. Therefore, testing is intrinsically an optimization problem: the tester tries to maximize the effectiveness of the test cases, at a minimum amount of resources.

This optimization problem is naturally modeled as a two-player game between the tester and the SUT: in each system state, the tester chooses one of the enabled input actions, and the SUT chooses an enabled output action. Thus, the inputs are under control of the tester, but the outputs of the SUT are not; which closely fits the nature of 2-player games.

Various authors have fruitfully pursued this idea and used game-based approaches to obtain effective testing strategies: this idea was first outlined in [13], and later refined in several ways, deploying Markov decision processes [12], timed games [9], and coverage games [20]. While they have shown their effectiveness in realistic settings, these methods are rather pragmatic in nature. A systematic comparison between model-based testing and the underlying game-theoretic methods is currently missing.

Indeed, existing game-based testing approaches formulate their methods in terms of a game graph between the tester and the SUT. Testers, however, do not think of testing as a game graph, but rather in terms of a system specification, from which tests are then derived. This means that earlier approaches leave implicit the step from the specification to the game graph. This step is crucial, since this is where the assumptions on the interaction between the tester and the SUT are encoded. In this paper, we show how different test assumptions are encoded by varying the move outcomes when the tester and the SUT propose their actions. In this way, we obtain a game based testing framework that is generic and overarches the most common testing assumptions.

Further, earlier approaches do not say how game strategies relate to test cases, hence it is not clear if game-based strategies and test cases have the same capabilities. In this paper, we show that each test case corresponds to a finite, trace based game strategy and vice versa, for history-dependent and deterministic

strategies that work on a slightly extended state space. Finally, we establish a fundamental connection between testing and games, by showing that game refinement corresponds to conformance. More precisely, we show that the so-called Player 2 alternating trace inclusion coincides with the canonical conformance relation in testing, namely the input-output conformance relation *ioco*.

One could argue that current game-based approaches to testing (including ours) focus on the competitive aspect of the interaction between two players: The tester plays *against* the SUT, so that the SUT makes the tester's life as difficult as possible. This means that winning strategies are too pessimistic: in reality, a tester and a SUT have a less hostile relationship. Therefore, an important topic for future work is to relax these assumptions, e.g. by deploying resilient strategies [7].

Contributions. This paper establishes a fundamental connection between model-based testing and 2-player concurrent games. In particular, we derive from a specification automaton a two player game, such that:

- *Test assumptions are explicitly encoded.* This makes our approach generic, so that game theoretic methods can be applied independent of the test assumptions.

- *Test cases correspond to game strategies* and therefore tests derived via classical test derivation algorithms, and game strategies derived via game theoretic algorithms have the same power.

- *Test case derivation corresponds to strategy synthesis*, so that any synthesis algorithm can be deployed to obtain (optimal) test cases.

- *Conformance coincides with alternating trace inclusion.* This provides a connection between two fundamental preorders.

More related work. As stated, various approaches to game-based test case deriviation exist. In particular, [12] uses $1\frac{1}{2}$-player games (i.e., Markov decision processes), to obtain cost-optimal test strategies, under the assumptions that the SUT behaves stochastically; [8, 9, 10] use timed games to find cost-optimal test cases for real-time games. In [3], games are used to generate test cases for Mealy machines based on CLT properties. Since Mealy machines are synchronous, the interaction between the tester and the SUT is much simpler, i.e. the players play in turns.

In [20], games are used to optimize state coverage, i.e., the number of (specification) states visited during testing; [6] studies the computational complexity of this problem. Finally, [1, 18] study the relation between game refinement (more precisely, alternating simulation), and the *ioco* conformance relation.

1 Games

We consider games played by two players on a game graph. In each state, both players choose one of their enabled actions, and together these determine the next states the game can be in. Since both actions are used simultaneously by a *Moves* function to decide on the next state, game arenas (Definition 1) describe concurrent 2-player games.

Definition 1. *A game arena is a tuple $G = (Q, q_0, Act_1, Act_2, \Gamma_1, \Gamma_2, Moves)$ where, for $i = 1, 2$:*

- *Q is a finite set of states,*
- *$q_0 \in Q$ is the initial state,*

- Act_i *is a finite and non-empty set of Player i actions,*

- $\Gamma_i : Q \to 2^{Act_i} \setminus \emptyset$ *is an enabling condition, which assigns to each state q a non-empty 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 $Moves(q,a,x) = \emptyset$ iff $a \notin \Gamma_1(q) \vee x \notin \Gamma_2(q)$.*

A *play* is an infinite path in a game arena, i.e., a sequence of states and actions of both players. We consider prefixes of plays as their finite description. A play is winning if it visits some state in a reachability goal $R \subseteq Q$.

Definition 2. *A* play *of a game arena $G = (Q, q_0, Act_1, Act_2, \Gamma_1, \Gamma_2, Moves)$ is an infinite sequence:*

$$\pi = q_0 \langle a_0, x_0 \rangle q_1 \langle a_1, x_1 \rangle q_2 \dots$$

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 $j \in \mathbb{N}$. We write $\pi_j^q \stackrel{\text{def}}{=} q_j$, $\pi_j^a \stackrel{\text{def}}{=} a_j$, and $\pi_j^x \stackrel{\text{def}}{=} x_j$ for the j-th state, player 1 action, and player 2 action respectively. The set of all plays of G is denoted $\Pi(G)$.

We define $\pi_{0:j} \stackrel{\text{def}}{=} q_0 \langle a_0, x_0 \rangle q_1 \langle a_1, x_1 \rangle q_2 \dots q_j$ as the prefix of play π up to the j-th state. With $|\pi|$ we denote the length *of a prefix π, i.e. the number of states in π. The set of all prefixes of a set of plays $P \subseteq \Pi(G)$ of G is denoted $Pref(P) \stackrel{\text{def}}{=} \{\pi_{0:j} \mid \pi \in P, j \in \mathbb{N}\}$. We define $\Pi^{pref}(G) \stackrel{\text{def}}{=} Pref(\Pi(G))$.*

Definition 3. *A play $\pi \in \Pi(G)$ of a game arena G is* winning *with respect to reachability goal $R \subseteq Q$, if π reaches some state in R, i.e., there exist a $j \in \mathbb{N}$ such that $\pi_j^q \in R$. We write $Win\Pi(G,R)$ for the set of winning plays with respect to R.*

The players will choose their actions for making a move according to a strategy (Definition 4). When the players execute their strategies in the game, we obtain a set of plays, called game outcomes. A strategy is winning if all the game outcomes are winning, no matter how the other player plays.

Definition 4. *A* strategy *for player i in game G is a function $\sigma_i : \Pi^{pref}(G) \to Act_i$, such that $\sigma_i(\pi) \in \Gamma_i(\pi_{|\pi|-1}^q)$ for any $\pi \in \Pi^{pref}(G)$. We write $\Sigma_i(G)$ for the set of all player i strategies in G. The* outcome *of two strategies $\sigma_1 \in \Sigma_1(G)$ and $\sigma_2 \in \Sigma_2(G)$ is the set of plays that occur when Player 1 plays according to σ_1 and Player 2 according to σ_2:*

$$Outc(\sigma_1, \sigma_2) = \{\pi \in \Pi(G) \mid \forall j \in \mathbb{N} : \sigma_1(\pi_{0:j}) = \pi_{j+1}^a \wedge \sigma_2(\pi_{0:j}) = \pi_{j+1}^x\}$$

The plays occurring for strategy $\sigma_1 \in \Sigma_1(G)$ are:

$$Outc(\sigma_1) \stackrel{\text{def}}{=} \bigcup \{Outc(\sigma_1, \sigma_2) \mid \sigma_2 \in \Sigma_2(G)\}$$

Strategy σ_1 is winning *with respect to reachability goal $R \subseteq Q$, if $Outc(\sigma_1) \subseteq Win\Pi(G,R)$. $Win\Sigma_1(G,R)$ denotes all winning player 1 strategies. Game G is* winning *for player 1 w.r.t. goal R iff $Win\Sigma_1(G,R) \neq \emptyset$.*

2   Model-based testing

Model-based testing is a smart way of testing that improves test efficiency by automatic test case generation, and execution, from a specification model, such that much manual (repetitive) labour can be

(a) Model-based testing

(b) SA specification of an MP3 player.
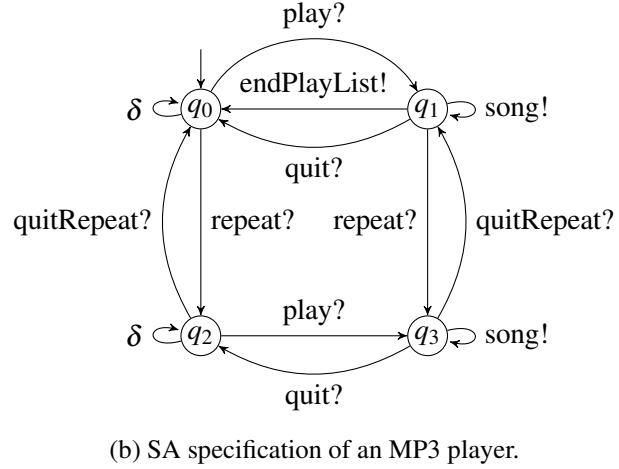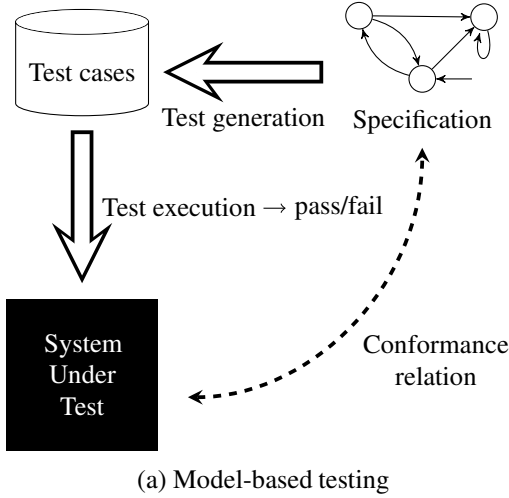
prevented. The specification is given as an automaton with input and outputs $\mathscr{A}$, that describes a restriction of desired system behavior. The goal of model-based testing is then to determine whether a given System Under Test (SUT) behaves as described by its specification, i.e. where the SUT *conforms* to $\mathscr{A}$.

Figure 1a illustrates the model-based testing process. A test case generation algorithm derives a set of test cases from the specification. Here, test cases are finite scenarios composed from inputs and outputs. Then, these test cases are executed automatically on the SUT. The SUT is considered as a *black box*, since we only see its input/output behavior, but not the internal workings, or source code. Finally, every test case is assigned a *test verdict*, being *Pass* or *Fail*. This verdict is determined in accordance with a conformance relation; in Section 5 we consider the input/output conformance relation *ioco* [17, 16].

2.1 Specifications as suspension automata

Following [4], we use suspension automata (SAs) as system specifications. These are determinised variants of the labeled transition systems with inputs and outputs from [17, 15].

For a partial function $f : X \rightharpoonup Y$, we write $f(x)\downarrow$ to denote that $f(x)$ is defined, and $f(x)\uparrow$ to denote that $f(x)$ is undefined.

Definition 5. *A suspension automaton (SA) is a 5-tuple $\mathscr{A} = (Q, L_I, L_O^\delta, T, q_0)$ where*

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

We write $L \stackrel{\text{def}}{=} L_I \cup L_O^\delta$. For $q \in Q$, we denote the set of enabled inputs and outputs in q by $in(q) = \{a \in L_I \mid T(q,a)\downarrow\}$, and $out(q) = \{x \in L_O^\delta \mid T(q,x)\downarrow\}$ respectively. We require that an SA is non-blocking: $\forall q \in Q : out(q) \neq \emptyset$.

We assume that any SA uses a special output label δ to indicate *quiescence*, i.e., the absence of an observable output $x \in L_O$ [17]. Handling quiescence is crucial in testing: if the SUT does not respond with any output, we must know whether or not this is allowed by the specification, otherwise we cannot come up with the correct verdict. We formalized this with the non-blocking requirement in Definition 5.

Example 1. *Figure 1b shows a specification of an MP3 player as an SA. In initial state q_0, no songs are played. Hence, this state has a self-loop labeled δ. After a* play? *input, the system moves to state q_1, in which songs are being played, until either* endPlayList! *occurs, or the* quit? *button is pressed. The MP3 player also features a repeat function, which can be switched on and off via the* repeat? *and* quitRepeat? *actions respectively. Thus, in state q_3, songs are played continuously, until the* quit? *action occurs.*

## 2.2   Test assumptions in case of input-output conflicts

SAs may feature states that enable both inputs and (non-quiescent) output actions. We call such a state *mixed*: formally, $q \in Q$ is mixed if $in(q) \neq \emptyset$ and $out(q) \neq \{\delta\}$. States q_1 and q_3 in Figure 1b are mixed. Mixed states may give rise to *input-output conflicts*. If the tester wants to take an input action, and the SUT wants at the same time to take an output action, the question arises which of the actions will be carried out. The literature introduced different ways to handle input/outputs conflicts, i.e. *test assumptions* on the interaction between the tester and the SUT. Note that there is no 'best' test assumption; this depends on the 'hostility' of the SUT against the tester. We list four test assumptions below.

- A test interaction is *input-eager (IE)* if the tester is always able to provide an input, even when the SUT wants to produce an output. This assumption prioritizes inputs over outputs and thereby makes mixed states fully controllable for the tester. This underlies the framework in [14].

- The converse of input-eager is an *output-eager (OE)* test interaction: the SUT always produces an output, unless δ is the only possible output. The authors of [11] use such an assumption. This assumption prioritizes outputs over inputs and thereby makes mixed states fully uncontrollable for the tester.

- A test interaction is *nondeterministic (ND)* if it is determined nondeterministically whether the SUT is able to take an output transition, or the tester to take an input transition in a mixed state. No guarantees are given on whether the tester is able to take an input transition in a mixed state, though this is not excluded as with the output-eager assumption. This assumption is similar to the test interaction used in the original theory for labeled transition systems with ioco [17].

- A test interaction is *input fair (IF)* if the tester is eventually able (after trying finitely many times) to take any input transition in a mixed state. Hence, mixed states are controllable for the tester, at the expense of trying multiple times. This assumption is made in [4].

In all test interactions of the assumptions above, either an input or an output action is ignored. One could also take both into account, by executing them both, but in nondeterministic order. This is a concurrent interpretation of an input-output conflict, which may be well suited for systems dealing with concurrent processes. If the SUT is able to receive more inputs than its specification specifies, this may lead to different interpretations of this assumption. Therefore, we will delay formalizing this assumption to future work. All test assumptions from the list above are formalized in Section 3, by incorporating the assumption in the *Moves* function from the underlying game arena of the specification.

Example 2. *An MP3 player, as described in Figure 1b, does not function properly, when taking the output-eager or nondeterministic test assumption. A real world MP3 player normally responds to input (though there may be some delay). With the printer from Figure 1c, we show that all test assumptions can impose a useful interpretation on the test interaction between the tester and the SUT. This specification models a printer which can handle printing and scanning in an interleaved way, i.e. printing does not need to be finished before scanning to be started and vice versa.*

- *The input-eager assumption allows the tester to always provide the inputs* print? *and* scan?*, before receiving outputs* printed! *and* scanned!*. Only if the tester decides to wait for an output, the SUT can produce these.*

- *The output-eager test assumption expresses that the specification is too complicated for the SUT that is being tested: the SUT cannot print and scan at the same time, because a* printed! *or* scanned! *output will occur after the tester has sent the respective input.*

- *With the nondeterministic test assumption, the tester may succeed in sending both inputs* print? *and* scan? *before receiving outputs, but no guarantees on success can be given. Furthermore, providing a* print? *input in mixed state q_1 may result in taking this transition, but instead, the* scanned! *output transition may also be taken.*

- *The input-fair test assumption is similar to the nondeterministic test assumption, but the difference is that it guarantees that the two input transitions* print? *and* scan? *can be taken from q_1 and q_3, before the outputs are produced, after trying a few times.*
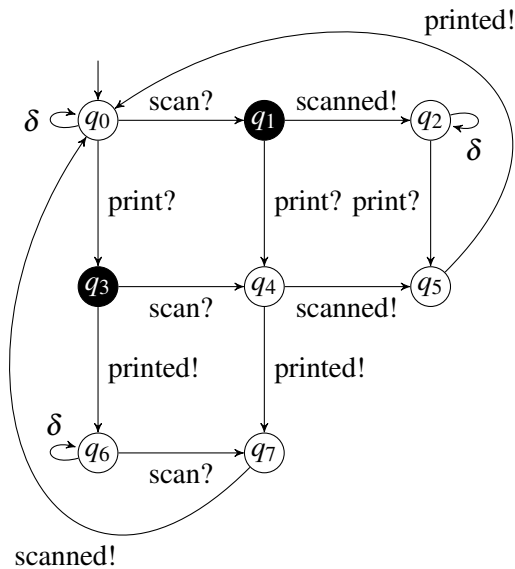
(c) SA specification of a printer. Mixed states are black.
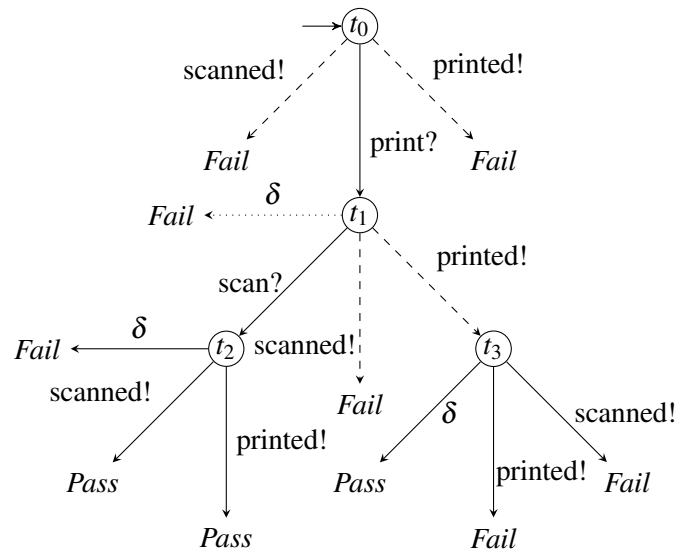
(d) Test case $\mathscr{T}$ of the SA of Figure 1c. For readability, *Pass* and *Fail* are displayed multiple times, and their self-loops have been omitted. See Example 3 for dotted and dashed transitions.

2.3 Test cases

We give a definition of test cases in the spirit of [17]. As shown in Figure 1d, a test case is a finite and acyclic SA $\mathscr{T}$. A test case is constructed by repeatedly taking either of the following test steps:

1. Choose an input a? from the input actions enabled in the current specification state, execute the input action, and move to the next state in $\mathscr{T}$. If the specification is in a mixed state, then it may happen that an output was observed before a? was observed. Therefore, if a? is enabled in a state of the test case, then all outputs from L_O are enabled as well.

2. Observe an output from the SUT. In case an output is observed that is prohibited by the specification, a fail verdict is emitted. Otherwise, one moves to the next state in $\mathscr{T}$.

3. Stop testing and emit a pass verdict. Note that all fail verdicts are handled in step 2.

Before defining test cases formally, we first define some auxiliary notation.

Definition 6. *Let $\mathscr{A} = (Q, L_I, L_O^\delta, T, q_0)$ be an SA, $q \in Q$, $Q' \subseteq Q$, $\mu \in L$, $\rho \in L^*$, and ε the empty sequence. Then we define:*

$$q \, after \, \varepsilon = \{q\} \qquad\qquad\qquad out(Q') = \bigcup_{q' \in Q'} out(q')$$

$$q \, after \, \mu\rho = \begin{cases} T(q,\mu) \, after \, \rho & if \, T(q,\mu)\!\downarrow \\ \emptyset & otherwise \end{cases} \qquad traces(\mathscr{A}) = \{\rho' \in L^* \mid \mathscr{A} \, after \, \rho' \neq \emptyset\}$$

$$\mathscr{A} \, after \, \rho = q_0 \, after \, \rho$$

Definition 7. *A test case for an SA $\mathscr{A}$ is an SA $\mathscr{T} = (Q^t, L_I, L_O^\delta, T^t, q_0^t)$ such that:*

- *There are two special states Pass, Fail$\in Q^t$ such that $T^t(Pass, x) = Pass$ and $T^t(Fail, x) = Fail$ for all $x \in L_O^\delta$, and $T^t(Pass, a) = T^t(Fail, a)\!\uparrow$ for $a \in L_I$.*

- *$\mathscr{T}$ has no cycles except those in Pass and Fail.*

- *Every state enables all outputs L_O, and either one input (matching step 1 above) or δ (matching step 2): $\forall q \in Q : (|in(q)| = 0 \wedge out(q) = L_O^\delta) \vee (out(q) = L_O \wedge |in(q)| = 1)$. Exception: in case of an input-eager test interaction, we only require: $\forall q \in Q : (|in(q)| = 0 \wedge out(q) = L_O^\delta) \vee |in(q)| = 1$.*

- *Traces of $\mathscr{T}$ leading to Pass, are traces of $\mathscr{A}$, while traces to Fail are not.*
  $\forall \rho \in traces(\mathscr{T}) : (\mathscr{T} \, after \, \rho = Pass \implies \rho \in traces(\mathscr{A})) \wedge (\mathscr{T} \, after \, \rho = Fail \implies \rho \notin traces(\mathscr{A}))$

Example 3. *Figure 1d shows a test case for the printer specification of Figure 1c. In state t_0, the tester provides the input* print?. *This state also has all non-quiescence outputs* printed! *and* scanned!, *because some test assumptions allow these to occur instead of the SUT accepting input* print?. *Since none of these actions are allowed by the specification in Figure 1c, these actions lead to a Fail verdict. In state t_1, the tester provides input* scan?, *but now one of the non-quiescent outputs is allowed, namely* printed!. *In state t_3, the tester decides to observe an output from the SUT. She may see either of three things: (1) quiescence (i.e. δ), leading to a Pass verdict, since quiescence is allowed in the specification, (2)* printed!, *which is not allowed, and (3)* scanned!, *which is also not allowed. After observing quiescence from state t_3, the tester decides to stop testing and conclude verdict Pass.*

Test cases are constructed without taking into account any specific test assumption, by including all inputs and outputs relevant for any of them. The dashed output transitions of t_0 and t_1 (and all of t_3) can be omitted in case of the input-eager test assumption. In case of the output-eager test assumption, input $\{$scan?$\}$ cannot be taken after $\{$print?$\}$, because specification state $T(q_0, \text{print?}) = q_3$ is mixed. One can adapt test case $\mathscr{T}$ to be relevant for the output-eager test assumption, by omitting the scan? *transition from t_1 (and all of t_2), while adding the dotted δ transition in t_1 for satisfying the third rule of Definition 7.*

3   Specifications are game arenas

To study the connection between test cases and games, we associate to each specification SA $\mathscr{A}$ a game arena $G_\mathscr{A}$. In $G_\mathscr{A}$, the tester (player 1) and the SUT (player 2) play on the state space given by $\mathscr{A}$, extended with a sink state $\bot$, and a number $i \in \{1, 2\}$ indicating whether the state was reached via a player 1 (input) or player 2 (output) action. The latter is important, because testers see the SUT via their traces, so we must record whose action was carried out.

To advance the game $G_\mathscr{A}$, both the tester and the SUT choose an action from the current state:

- The tester chooses either an enabled input from the specification SA $\mathscr{A}$, or one of the special inputs θ and stop?. The θ action expresses that the tester desires to take no input, and allows the SUT to execute any output he wishes; the stop? action indicates that the tester wants to stop testing, which brings the game to the state $\perp$.

- The SUT chooses one of the enabled outputs from the specification.

Then the game moves to a next state, according the function *Moves*, which reflects how the tester and the SUT interact. Hence, different test assumptions made in the literature give rise to different definitions of the *Moves* function.

The explicit game definition (Definition 8) and encoding of test assumptions (Section 3.1) set our work apart from earlier game-based approaches to testing. In particular, [6] derive optimal test cases from a specification that is already given as a game. The encoding also allows us to study the relation between test cases and games strategies (Section 4), and between alternating refinement and conformance (Section 5).

Definition 8. *Let $\mathscr{A} = (Q, L_I, L_O^\delta, T, q_0)$ be an SA. The* game arena underlying $\mathscr{A}$ *is defined by $G_\mathscr{A} = (Q_\perp, (q_0, 1), Act_1, Act_2, \Gamma_1, \Gamma_2, Moves)$ where:*

- $Q_\perp = (Q \times \{1, 2\}) \cup \{(\perp, 1)\}$
- $Act_1 = L_I \cup \{\theta, \text{stop?}\}$ and $Act_2 = L_O^\delta$,
- *for all $q \in Q$ and $i \in \{1, 2\}$, we take $\Gamma_1((q, i)) = in(q) \cup \{\theta, \text{stop?}\}$ and $\Gamma_2((q, i)) = out(q)$,*
- *we take $\Gamma_1(\perp, 1) = \{\text{stop?}\}$ and $\Gamma_2(\perp, 1) = L_O^\delta$, and*
- *the function Moves : $Q_\perp \times Act_1 \times Act_2 \to 2^{Q_\perp}$ encodes one of the different test assumptions and is given in Subsection 3.1. Besides the requirement from Definition 1 for moves with undefined action, we require $Moves(q, \text{stop?}, x) = (\perp, 1)$, and $Moves((\perp, 1), \text{stop?}, x) = (\perp, 1)$.*

For the remainder of the paper, we fix a specification $\mathscr{A}$, and its underlying game $G_\mathscr{A}$.

3.1 Encoding test assumptions

We formalize test interaction by implementing a *Moves* function for each of the test assumptions. Note that the *Moves* function is of type $Q_\perp \times (L_I \cup \{\theta, \text{stop?}\}) \times L_O^\delta \to 2^{Q_\perp}$, i.e. it takes a game state, an input from the tester, and an output from the SUT. The function returns a set of the reached next states for the given input and output, using the transition function of the SA for which this *moves* function is defined.

All the *Moves* functions from Definition 9 use the symbols δ and θ as special symbols that transfer control to the tester or SUT respectively. The symbol δ is used in its usual semantics, i.e. it denotes quiescence. When the SUT is quiescent, the tester is always able to provide an input. However, δ can only actually be observed if the tester chooses the θ input. Hence, θ is then used as an artificial input to model that the tester is waiting for the SUT to take an output transition. This corresponds with how θ is used in [17].

In practice, δ can be observed by setting a timeout. It is then assumed that the SUT does not produce any regular output from L_O after this timeout. Input θ can be implemented in practice, by waiting for the SUT to produce an output.

The behaviour of the *Moves* functions from Definition 9 differ for the regular inputs $in(q)$ and outputs $out(q) \setminus \{\delta\}$ of a state q of the SA, in order to resolve input/output conflicts according to one of the test assumptions from Section 2. We will discuss how to implement the input-fair test assumption in Section 3.2, because its semantics cannot be implemented directly with only a *Moves* function.

Definition 9. *Let $\mathscr{A} = (Q, L_I, L_O, T, q_0)$ be an SA. The various test assumptions from Section 2 give rise to the following functions Moves : $Q_\perp \times (L_I \cup \{\theta, \text{stop?}\}) \times L_O^\delta \to 2^{Q_\perp}$ (we do not include the moves from Definition 8 for $(\perp, q)$, stop? and undefined actions here):*

- *In the input-eager regime, an input $a \in L_I$ is always executed, unless the tester decides to not perform an input, i.e. she proposes $a = \theta$.*

$$Moves_{IE}((q,i),a,x) = \begin{cases} \{(T(q,a),1)\} & \text{if } a \neq \theta \\ \{(T(q,x),2)\} & \text{otherwise} \end{cases}$$

- *In the output-eager regime, the output action $x \in L_O$ is always executed, unless both $x = \delta$ and $a \neq \theta$.*

$$Moves_{OE}((q,i),a,x) = \begin{cases} \{(T(q,x),2)\} & \text{if } x \neq \delta \ \vee \ a = \theta \\ \{(T(q,a),1)\} & \text{otherwise} \end{cases}$$

- *In the nondeterministic regime, a nondeterministic choice is made whether to execute the input (unless $a = \theta$) or the output (unless $x = \delta$ and $a \neq \theta$). We take the same Moves function for the input-fair regime, and explain this in Section 3.2.*

$$Moves_{ND}((q,i),a,x) = \begin{cases} \{(T(q,x),2)\} & \text{if } a = \theta \\ \{(T(q,a),1)\} & \text{if } x = \delta \\ \{(T(q,a),1),(T(q,x),2)\} & \text{otherwise} \end{cases}$$

Testing with a hard reset.   Further, it is often useful to have a hard reset function. In testing, a reset function is used to execute multiple test cases from the initial state. In the specification, a hard reset is enabled in any state, and the tester can always use it to go back to the initial state, no matter whether the SUT wanted to do an output action. In practice, a SUT often needs to be instrumented for implementing such a reset function in a fast way, as rebooting a system can take a lot of time. In Definition 10, we have adapted Definition 8, to include this special input *reset?* $\notin L_I$.

Definition 10. *Let $\mathscr{A} = (Q, L_I, L_O^\delta, T, q_0)$ be an SA. The resettable game arena underlying $\mathscr{A}$ is defined by $G_{\mathscr{A}}^{reset?} = (Q_\perp, (q_0, 1), Act_1, Act_2, \Gamma_1, \Gamma_2, Moves)$ where:*

- $Q_\perp = (Q \times \{1,2\}) \cup \{(\perp,1)\}$
- $Act_1 = L_I \cup \{\theta, \text{stop?}, \text{reset?}\}$ and $Act_2 = L_O^\delta$
- *for all $q \in Q$ and $i \in \{1,2\}$, we take $\Gamma_1((q,i)) = in(q) \cup \{\theta, \text{stop?}, \text{reset?}\}$ and $\Gamma_2((q,i)) = out(q)$,*
- *we take $\Gamma_1(\perp,1) = \{\text{stop?}\}$ and $\Gamma_2(\perp,1) = L_O^\delta$, and*
- *the function Moves : $Q_\perp \times Act_1 \times Act_2 \to 2^{Q_\perp}$ encodes one of the test assumptions from Definition 9, with the modification that $\forall q \in Q, \forall i \in \{1,2\}, \forall x \in \Gamma_2((q,i)) : Moves((q,i),\text{reset?},x) = (q_0,1)$.*

3.2   Encoding the input-fair test assumptions via fair plays

To define the input-fair test assumption from Section 2, we introduce the notion of a *fair play*. In essence, a play π is input fair if, whenever the tester wants to perform an input action from state q, it is eventually executed in q [4]. Given a state q and an input action a that is enabled in q, we say that a play is *fair with respect to q and a* if a is actually taken in q at some point in time. A play π is *fair* if it is fair with respect to any q appearing in π and any input action that was proposed in q in π.

Definition 11. *Let $\pi \in \Pi(G_{\mathscr{A}})$. Then π is* input-fair *w.r.t. some state $q \in Q$ and some input $a \in in(q)$ if:*

$$\exists j \in \mathbb{N}, i \in \{1,2\}, q' \in Q : \pi_j^a = a \wedge \pi_j^q = (q,i) \wedge \pi_{j+1}^q = (q',1)$$

Play π is input-fair *if:*

$$\forall q \in Q, a \in L_I : (\exists j \in \mathbb{N}, i \in \{1,2\} : \pi_j^a = a \wedge \pi_j^q = (q,i)) \implies \pi \text{ is input-fair w.r.t. } q \text{ and } a$$

Clearly, restricting an underlying game arena of an SA with the nondeterministic test assumption to input-fair plays only results in satisfying the description of the input-fair test assumption from Section 2. Instead of restricting the plays of the game, we will leave the game with the nondeterministic test assumption as is, and consider input-fair strategies in Definition 12.

Definition 12. *Let $G_{\mathscr{A}}^{ND}$ be the underlying game arena of SA $\mathscr{A}$ with the nondeterministic Moves function from Definition 9. Then a strategy $\sigma \in \Sigma_i(G_{\mathscr{A}})$ is* input-fair *if it is winning on the set of all input-fair plays of G_A.*

Requiring input-fairness may appear restrictive for specifications with states that cannot be reached in any way once leaving them via some transition. However, adding a reset to the game (Definition 10) resolves this, as a state reached by some play, can then always be reached again by resetting, and following the same play. Of course, the SUT may also be able to prevent the tester from following this play right away, but input-fairness ensures that this is possible after trying a few times.

3.3 Comparing strategies for different test assumptions

This section compares strategies for game arenas based on the different test assumptions, constructed from the same specification. In particular, we show that, for any reachability objective R, either both the input-eager and input-fair games are winning for player 1, or both not. Similarly, for objective R, either both the nondeterministic and output-eager games are winning for player 1 or both not.

To show that the input-eager and input-fair games coincide, we observe that any winning player 1 strategy of an input-fair game is also winning in an input-eager game. The converse does not hold however, because the set of plays of an input-eager game has fewer plays than the input-fair game, as the input-fair game does not have plays in which a proposed input transition is not taken. However, by reaching the same state multiple times, we know that the winning action proposed in the input-eager game is eventually carried out.

A similar reasoning holds for the winning player 1 strategies of games based on the nondeterministic and output-eager test assumption. There are no guarantees for taking an input transition in a mixed state (according to the nondeterministic test assumption), or it is simply not possible (according to the output-eager test assumption). Hence, in both cases, a winning strategy cannot make use of taking input transitions in mixed states. Hence, the existence of winning strategies is the same in both games.

Theorem 1. *Let $\mathscr{A} = (Q, L_I, L_O, T, q_0)$ be an SA, and $G_{\mathscr{A}}^{IE}, G_{\mathscr{A}}^{IF}, G_{\mathscr{A}}^{OE}, G_{\mathscr{A}}^{ND}$ the underlying game arenas of $\mathscr{A}$ for the input-eager, input-fair, output-eager, and nondeterministic test assumption, respectively. Let $R \subseteq Q$ be a reachability goal. Then:*

1. *$G_{\mathscr{A}}^{IE}$ is winning for player 1 w.r.t. R if and only if $G_{\mathscr{A}}^{IF}$ is winning for player 1 w.r.t. R, and*

2. *$G_{\mathscr{A}}^{OE}$ is winning for player 1 w.r.t R if and only if $G_{\mathscr{A}}^{ND}$ is winning for player 1 w.r.t. R.*

4   Test cases are game strategies (and vice versa)

This section establishes a strong correspondence between Player 1 strategies and test cases. To achieve this result, we observe that test cases and strategies share many features, but differ on two aspects: (1) Test cases are finite, while strategies play forever. Hence, test cases correspond to Player 1 strategies that are finite, i.e., eventually provide a stop? action. (2) Test cases base their decisions on the observed traces only, while strategies can use all information contained in the plays, especially the proposed actions for which the corresponding transition was not taken. Therefore, test cases correspond to finite trace-based Player 1 strategies. Thus, we establish a bijection between test cases and finite, trace-based strategies. We first explain how a strategy can be extracted from a test case, and then how a test case can be extracted from a strategy.

4.1   From strategies to test cases

We derive a test case from each finite and trace-based Player 1 strategy (Definition 13). A strategy is *trace-based* if its choices only depend on the observed traces. Finite strategies eventually provide a stop? action. Note that, after one stop? action from Player 1, all subsequent Player 1 actions are stop?. Hence, only the prefix before the stop?-action matters (Definition 14).

Exactly the traces of these prefixes cut off before the stop? actions can be used to construct a test case. Each of these traces either leads to the *Pass* state, or to a unique test case state. A test case only consisting of these traces does not contain any output transition to the *Fail* state, because these outputs do not occur in the specification. Hence, they need to be added. This construction proves Theorem 2.

Definition 13. *The function trace* $: \Pi^{pref}(G_{\mathscr{A}}) \to L^*$ *assigns to each play prefix* $\pi = (q_0, i_0)\langle a_0, x_0\rangle(q_1, i_1)$ $\dots \langle a_{n-1}, x_{n-1}\rangle(q_n, i_n)$, *a sequence of action labels* $\rho = \rho_0\rho_1, \dots \rho_{n-1}$ *given by* $\rho_j = a_j$ *if* $i_{j+1} = 1$ *and* $\rho_j = x_j$ *otherwise. A strategy* $\sigma \in \Sigma_i(G_{\mathscr{A}})$ *is called* trace-based *if:*

$$\forall \pi, \pi' \in \Pi^{pref}(G) : trace(\pi) = trace(\pi') \implies \sigma(\pi) = \sigma(\pi')$$

A strategy $\sigma_1 \in \Sigma_1(G_{\mathscr{A}})$ *is* finite *if:*

$$\forall \pi \in Outc(\sigma_1), \exists j \in \mathbb{N} : \pi_j^a = \text{stop?}$$

Example 4. *A strategy is not trace-based, if it returns different actions for a play that consists of the same executed actions that form the trace, and different non-executed actions. This situation cannot occur for the printer from Figure 1c, so we give an example player 1 strategy for the MP3 player of Figure 1b:*

$$\sigma_1(\pi) = \begin{cases} \text{play?} & \text{if } \pi = (q_0, 1)\langle\text{play?}, \delta\rangle(q_1, 1)\langle\text{quit?}, \text{endPlayList!}\rangle(q_0, 1) \\ \text{repeat?} & \text{if } \pi = (q_0, 1)\langle\text{play?}, \delta\rangle(q_1, 1)\langle\text{quit?}, \text{song!}\rangle(q_0, 1) \\ \theta & \text{otherwise} \end{cases}$$

State q_1 *enables two outputs (namely* song! *and* endPlayList!*) which are both not executed, in the two plays mentioned above, because the input transition for* quit? *has been taken (as indicated by the 1 in the last state). Nevertheless,* σ *returns either* play? *or* repeat? *based on the non-executed outputs.*

Definition 14. *Let* $\sigma_1 \in \Sigma_1(G_{\mathscr{A}})$ *be a player 1 trace-based, finite strategy in* $G_{\mathscr{A}}$. *We define a trace set* $T_{\sigma_1} = \{trace(\pi) \mid \pi \in Pref(Outc(\sigma_1)) \wedge \pi_{|\pi|-1}^a \neq \text{stop?}\}$.

Theorem 2. *Then* T_{σ_1} *characterizes a unique test case.*

Example 5. *Let $\sigma_1 \in \Sigma_1(G_{\mathscr{A}})$ be defined as follows:*

$$\sigma_1(\pi) = \begin{cases} \text{stop?} & \text{if } |\pi| \geq 4 \\ \text{print?} & \text{if } |\pi| < 4 \wedge \text{print?} \in \Gamma_1(\pi^q_{|\pi|-1}) \\ \text{scan?} & \text{if } |\pi| < 4 \wedge \text{print?} \notin \Gamma_1(\pi^q_{|\pi|-1}) \wedge \text{scan?} \in \Gamma_1(\pi^q_{|\pi|-1}) \\ \theta & \text{otherwise} \end{cases}$$

Note that strategy σ_1 is finite and trace-based. The trace set of σ_1 is $T_{\sigma_1} = \{\text{print?}, \text{print?printed!}, \text{print?printed!}\delta, \text{print?scan?}, \text{print?scan?printed!}, \text{print?scan?scanned!}\}$, in case of the input-fair or nondeterministic test assumption. This set is exactly the prefix-closed set of the traces leading to a Pass state in the test case of Figure 1d. Note that if $G_{\mathscr{A}}$ uses the input-eager test assumption, traces print?printed! and print?printed!δ are not included in $\mathscr{T}_{\sigma_1}$. The traces print?scan?, print?scan?printed!, and print?scan?scanned! are not included in $\mathscr{T}_{\sigma_1}$ in case of the output-eager test assumption.

4.2 From test cases to strategies

Given a test case $\mathscr{T}$ for an SA $\mathscr{A}$, we construct a game strategy σ_1 as follows. On play prefixes π whose traces are included in $\mathscr{T}$, σ_1 returns the input action enabled in the state of $\mathscr{T}$ reached by this trace, if it has one. If π has a trace in $\mathscr{T}$ leading to (or passing by) the *Pass* state, then σ_1 returns the action stop?. In all other cases, we set $\sigma_1(\pi) = \theta$, because the trace of π then either reaches a state of $\mathscr{T}$ with no enabled input transition, or π is a play that does not occur in the outcome of the game when using σ_1.

Definition 15. *Let $\mathscr{T}$ be a test case for an SA $\mathscr{A}$. We define a strategy $\sigma_{\mathscr{T}}$ of $G_{\mathscr{A}}$ as follows:*

$$\sigma_{\mathscr{T}}(\pi) = \begin{cases} a & \text{if } \exists q \in Q, a \in L_I : \mathscr{T} \text{ after } (trace(\pi)) = \{q\} \wedge in(q) = \{a\} \\ \text{stop?} & \text{if } \exists j \in \mathbb{N} : Pass \in \mathscr{T} \text{ after } (trace(\pi_{0:j})) \\ \theta & \text{otherwise} \end{cases}$$

Note that $\sigma_{\mathscr{T}}$ is well-defined, because by Definition 7, the input a in the first clause is unique, if it exists. Theorem 3 then states that $\sigma_{\mathscr{T}}$ is finite and trace-based Player 1 strategy. Further, σ_T is unique, i.e. from a test case exactly one strategy can be derived.

Theorem 3. *Let $\mathscr{T}$ be a test case for SA $\mathscr{A}$. Then we have*

1. *The strategy $\sigma_{\mathscr{T}}$ is a Player 1 strategy in $G_{\mathscr{A}}$.*

2. *$\sigma_{\mathscr{T}}$ is finite and trace-based.*

3. *If $\sigma_{\mathscr{T}} = \sigma_{\mathscr{T}'}$ then $\mathscr{T} = \mathscr{T}'$.*

Example 6. *We use Definition 14 to construct the following strategy from test case $\mathscr{T}$ of Figure 1d:*

$$\sigma_{\mathscr{T}}(\pi) = \begin{cases} \text{print?} & \text{if } trace(\pi) = \varepsilon \\ \text{scan?} & \text{if } trace(\pi) = \text{print?} \\ \text{stop?} & \text{if } trace(\pi) \in \{\text{print?scan?printed!stop?}^*, \text{print?scan?scanned!stop?}^*, \\ & \qquad\qquad\qquad\qquad\qquad\qquad \text{print?printed!}\delta\text{stop?}^*\} \\ \theta & \text{otherwise} \end{cases}$$

Note that this strategy is equivalent to the one from Example 5.

4.3   Test case generation is strategy synthesis

We can now establish that we have defined a bijective function between test cases and strategies, by using the translation from strategy to test case from Theorem 2, and the translation from test case to strategy from Theorem 3 as its inverse.

Theorem 4. *The function $\mathscr{T} \mapsto \sigma_{\mathscr{T}}$ is a bijection from the set of test cases of $\mathscr{A}$ to the set of finite trace-based strategies of $G_{\mathscr{A}}$.*

A consequence of Theorem 4 is that game synthesis algorithms can be used for deriving test cases for specific *test objectives*. Test objectives describe the objective that a tester likes to achieve during testing.

Various test objectives exist: *Reachability goals* [5] are states in the specification that the tester likes to reach. For example, one may like to see that the MP3 player is able to play songs. To do so, the tester likes to reach any state with an outgoing song-transition (and see if this transition can be executed), conforming to states q_1 and q_3 in Figure 1b. Test purposes [19] generalize reachability goals in the sense that a whole scenario needs to be executed; for example, one likes to see if the MP3 player can produce songs, after a quit? action. Since such scenarios can be adaptive, we model a test purpose as an SA with final states, in which the test purpose was successfully executed. This idea is common in model-based testing [19], but has not been exploited in a game theoretic setting. The interaction between the specification and the test purpose is modeled via a composition operator $\|$. Finally, (state) *coverage* [6] can be a test objective, where the tester tries to cover as many states in the specification as possible. As stated, our framework enables strategy synthesis for these test objectives for any test regime.

5   Conformance is alternating trace inclusion

A popular conformance relation for model-based testing is *input-output conformance*, *ioco* for short [17]. This relation formalizes what it means that an SUT, modeled as an input-enabled suspension automaton *Impl*, conforms to a specification, modeled as an SA *Spec* (Definition 16); an SA $\mathscr{A}$ is input-enabled if all its states q satisfy $in(q) = L_I$. The *ioco* relation allows the implementation to implement more inputs, and fewer outputs than the specification. Indeed, implementation *Impl* may implement more services than specified in *Spec*, but on the specified inputs it must behave as prescribed by *Spec*.

This viewpoint corresponds to Player 2 alternating trace inclusion $\sqsubseteq_2$ for games [2]. Game G_A is 2-alternating-trace included in game G_B, if any trace set that can be enforced by Player 2 in G_A can also be enforced by Player 2 in G_B (Theorem 5).

In the definition of alternating trace inclusion (Definition 18), player 1 chooses an input in $G_{\mathscr{B}}$, and then needs to choose a corresponding input in $G_{\mathscr{A}}$. However, we need to take care that player 1 does not cheat in $G_{\mathscr{A}}$, by choosing the θ input, if this input is not chosen in $G_{\mathscr{B}}$ (Definition 17).

Definition 16. *Let $\mathscr{A}$ and $\mathscr{B}$ be SAs over the same label sets and assume that $\mathscr{A}$ is input-enabled. Then we say that $\mathscr{A}$ ioco $\mathscr{B}$ if for all $\rho \in traces(\mathscr{B})$ we have $out(\mathscr{A} \ after \ \rho) \subseteq out(\mathscr{B} \ after \ \rho)$.*

Definition 17. *Let G be a game arena corresponding to some SA, and $\pi = (q_0, j_0)\langle a_0, x_0 \rangle (q_1, j_1)\langle a_1, x_1 \rangle (q_2, j_2) \ldots \langle a_{k-1}x_{k-1} \rangle (q_k, j_k) \in \Pi^{pref}(G)$ a play prefix of G. Then the action decision sequence of π is:*

$$actions(\pi) \stackrel{\text{def}}{=} j_0 \langle a_0, x_0 \rangle j_1 \langle a_1, x_1 \rangle j_2 \ldots \langle a_{k-1}x_{k-1} \rangle j_k$$

Let $G_{\mathscr{A}}, G_{\mathscr{B}}$ be two game arenas corresponding to an input-enabled SA $\mathscr{A}$, and an SA $\mathscr{B}$, respectively. Let $\sigma_1^A \in \Sigma_1(G_{\mathscr{A}})$, $\sigma_1^B \in \Sigma_1(G_{\mathscr{B}})$ be two player 1 strategies in these games. Strategy σ_1^A cheats on σ_1^B if:

$$\exists \pi \in \Pi^{pref}(G_{\mathscr{A}}), \forall \pi' \in \Pi^{pref}(G_{\mathscr{B}}) : actions(\pi) = actions(\pi') \wedge \sigma_1^B(\pi') \neq \theta \implies \sigma_1^A(\pi) = \theta$$

Definition 18. *Let G_A and G_B be game arenas corresponding to an input-enabled SA $\mathscr{A}$, and an SA $\mathscr{B}$, respectively. We say that G_A is alternating trace included in G_B, denoted $G_A \sqsubseteq_2 G_B$ iff*

$$\forall \sigma_2^A \in \Sigma_2(G_A), \exists \sigma_2^B \in \Sigma_2(G_B), \forall \sigma_1^B \in \Sigma_1(G_B), \exists \sigma_1^A \in \Sigma_1(G_A) : \sigma_1^A \text{ does not cheat on } \sigma_1^B, \text{ and}$$

$$\{trace(\pi) \mid \pi \in Pref(Outc(\sigma_1^A, \sigma_2^A))\} \subseteq \{trace(\pi) \mid \pi \in Pref(Outc(\sigma_1^B, \sigma_2^B))\}$$

Theorem 5. *Let $\mathscr{A}$ and $\mathscr{B}$ be SAs over the same label sets and assume that $\mathscr{A}$ is input-enabled. Let $G_{\mathscr{A}}$ and $G_{\mathscr{B}}$ be their respective underlying game arenas for the nondeterministic test assumption. Then:*

$$\mathscr{A} \text{ ioco } \mathscr{B} \iff G_{\mathscr{A}} \sqsubseteq_2 G_{\mathscr{B}}$$

The relation between game refinement and ioco has been studied before: [1, 18] show that, on interface automata, *ioco* corresponds to alternating simulation. Theorem 5 differs from these results in three ways: (1) [1, 18] compare ioco and alternating simulation on interface automata. We compare ioco on SAs versus alternating trace inclusion on games, (2) Our games consider concurrent moves by both players; interface automata compare different transitions of the same player. (3) Alternating trace inclusion is linear, whereas alternating simulation is a branching time relation. One could however argue that simulation and trace inclusion coincide for deterministic systems (including our SAs). However, we prefer the formulation in terms of alternating traces, because we conjecture that this formulation extends to the non-deterministic case.

6   Conclusions and Future Work

We have established a fundamental connection between model-based testing and 2-player concurrent games, where specifications are game arenas, test cases are game strategies, test case derivation is strategy synthesis, and conformance is alternating trace inclusion. This connection allows the wide plethora of game synthesis techniques to be deployed to test case generation.

The game theoretic setting spawns several game theoretic questions. While the games we propose are concurrent because both the tester and the SUT propose moves at the same time, one could argue that they are only semi-concurrent, since only one of these moves is carried out at the same time. Therefore, we believe that the test games have various properties that do not hold for concurrent games in general. We conjecture that, whereas concurrent games require probabilistic strategies to win reachability properties, our games require only deterministic strategies to win these, and we also believe that our games are determined in that case.

Proofs   For proofs of the theorems in this paper, we refer the reader to `https://petravdbos.nl/`

References

[1] Fides Aarts & Frits Vaandrager (2010): *Learning I/O Automata*. In: *International Conference on Concurrency Theory*, Springer, pp. 71–85, doi:10.1007/978-3-642-15375-4_6.

[2] Rajeev Alur, Thomas Henzinger, Orna Kupferman & Moshe Vardi (1998): *Alternating Refinement Relations*. In: *International Conference on Concurrency Theory*, Springer, pp. 163–178, doi:10.1007/BFb0055622.

[3] Roderick Bloem, Robert Könighofer, Ingo Pill & Franz Röck (2016): *Synthesizing Adaptive Test Strategies from Temporal Logic Specifications*. In: *Formal Methods in Computer-Aided Design*, IEEE, pp. 17–24, doi:10.1109/FMCAD.2016.7886656.

[4] Petra van den Bos, Ramon Janssen & Joshua Moerman (2017): *n-Complete Test Suites for IOCO*. In: *IFIP International Conference on Testing Software and Systems*, Springer, pp. 91–107, doi:10.1007/978-3-319-67549-7 6.

[5] Laura Brandán Briones & Hendrik Brinksma (2004): *A Test Teneration Framework for Quiescent Real-Time Systems*. In: *Proc. Formal Approaches to Testing of Software (4th International Workshop)*, pp. 71 – 85, doi:10.1007/978-3-540-31848-4 5.

[6] Krishnendu Chatterjee, Luca De Alfaro & Rupak Majumdar (2008): *The Complexity of Coverage*. In: *Asian Symposium on Programming Languages and Systems*, Springer, pp. 91–106, doi:10.1007/978-3-540-89330-1 7.

[7] Eric Dallal, Daniel Neider & Paulo Tabuada (2016): *Synthesis of Safety Controllers Robust to Unmodeled Intermittent Disturbances*. In: *Decision and Control (CDC), 2016 IEEE 55th Conference on*, IEEE, pp. 7425–7430, doi:10.1109/CDC.2016.7799416.

[8] Alexandre David, Kim Larsen, Shuhao Li & Brian Nielsen (2008): *Cooperative Testing of Timed Systems*. *Electronic Notes in Theoretical Computer Science*, pp. 79–92, doi:10.1016/j.entcs.2008.11.007.

[9] Alexandre David, Kim Larsen, Shuhao Li & Brian Nielsen (2008): *A Game-Theoretic Approach to Real-Time System Testing*. In: *Design, Automation and Test in Europe*, IEEE, pp. 486–491, doi:10.1145/1403375.1403491.

[10] Alexandre David, Kim Larsen, Shuhao Li & Brian Nielsen (2009): *Timed Testing under Partial Observability*. In: *International Conference on Software Testing Verification and Validation*, IEEE, pp. 61–70, doi:10.1109/ICST.2009.38.

[11] Niklas Krafczyk & Jan Peleska (2017): *Effective Infinite-State Model Checking by Input Equivalence Class Partitioning*. In: *IFIP International Conference on Testing Software and Systems*, Springer, pp. 38–53, doi:10.1007/978-3-319-67549-7 3.

[12] Lev Nachmanson, Margus Veanes, Wolfram Schulte, Nikolai Tillmann & Wolfgang Grieskamp (2004): *Optimal Strategies for Testing Nondeterministic Systems*. *ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 55–64, doi:10.1145/1013886.1007520.

[13] Christos Papadimitriou (2001): *Algorithms, Games, and the Internet*. In: *Proceedings of the thirty-third annual ACM symposium on Theory of computing*, ACM Press, pp. 749–753, doi:10.1145/380752.380883.

[14] Adenilso Simao & Alexandre Petrenko (2014): *Generating Complete and Finite Test Suite for ioco: Is It Possible?* In: *Proceedings of the Ninth Workshop on Model-Based Testing*, pp. 56–70, doi:10.4204/EPTCS.141.5.

[15] Willem Stokkink, Mark Timmer & Mariëlle Stoelinga (2013): *Divergent Quiescent Transition Systems*. In: *Proceedings seventh conference on Tests and Proofs*, LNCS, doi:10.1007/978-3-642-38916-0 13.

[16] Mark Timmer, Hendrik Brinksma & Mariëlle Stoelinga (2011): *Model-Based Testing*. In: *Software and Systems Safety: Specification and Verification*, NATO Science for Peace and Security, IOS Press, pp. 1–32, doi:10.3233/978-1-60750-711-6-1.

[17] Jan Tretmans (2008): *Model Based Testing with Labelled Transition Systems*. In: *Formal methods and testing*, Springer, pp. 1–38, doi:10.1007/978-3-540-78917-8 1.

[18] Margus Veanes & Nikolaj Bjørner (2010): *Alternating Simulation and IOCO*. In: *IFIP International Conference on Testing Software and Systems*, Springer, pp. 47–62, doi:10.1007/978-3-642-16573-3 5.

[19] René de Vries & Jan Tretmans (2001): *Towards Formal Test Purposes*. *Formal Approaches to Testing of Software, FATES'01: A Satellite Workshop of CONCUR'01 Proceedings*, pp. 61–76.

[20] Farn Wang, Sven Schewe & Jung-Hsuan Wu (2015): *Complexity of Node Coverage Games*. *Theoretical Computer Science*, pp. 45–60, doi:10.1016/j.tcs.2015.02.002.