Formalising Concurrent Test Interactions in Model-based Testing via Games

KARSTEN E. ROELOFS, University of Twente, The Netherlands

In preceding research, a strong correlation between model-based testing and game theory has been established and formalised. In this conversion, an assumption must be made on which transition in a game arena to take when tester and system propose conflicting actions. Several such assumptions exist, but always discard and ignore either of these actions. We investigate a conjecture from earlier literature concerning a novel concurrent test assumption taking into account both. We formalise the assumption and evaluate its efficacy. Based on our findings we conclude that the novel test assumption may allow for the accurate modeling of concurrent software systems, in addition to possibly providing an increase in overall testing efficiency. We identify multiple avenues for further research.

Additional Key Words and Phrases: model-based testing, game theory, concurrent software systems

1 INTRODUCTION

In the world of software engineering, a process of vital importance is the constant and rigorous testing of the system being developed. To ensure that this system under test (SUT) conforms to its specification, and to gain insight into its possible faults, it is subjected to a large number of test cases. The process of establishing these test cases for a given system has classically been, both time-wise and monetarily, an expensive endeavour, often claiming significant parts of development budgets (estimated at 30-50% [8]). Evidently, much benefit would be gained from tools which may facilitate cheaper and less labour intensive testing of systems.

The field of model-based testing contributes to this effort by attempting to automate the creation of test cases. It introduces methods to create formal specifications of software systems, and gives algorithms to subsequently generate test cases. Several methods for model-based test case generation exist [3, 4, 7] which have enabled testers to automate software testing and save resources.

Other authors [5, 2, 9] have found that results could be achieved by interpreting a specification model as a game arena, and subsequently applying game-theoretical strategy synthesis techniques to determine test cases. This connection between testing and games has been formalised into a generic framework by Van den Bos & Stoelinga [1], where test cases were shown to correspond to game strategies and test case derivation was shown to correspond to strategy synthesis. A schematic overview of this process is shown in Figure 1.

When considering model-based testing, one has to decide what is to happen in the event that the tester wishes to provide an input at the same time that the SUT produces an output. To solve these so called *input-output conflicts*, the literature has introduced several distinct *test assumptions* prescribing how the test interactions between the SUT and tester (i.e., the series of proposed and executed or ignored actions over a period of time) should look.

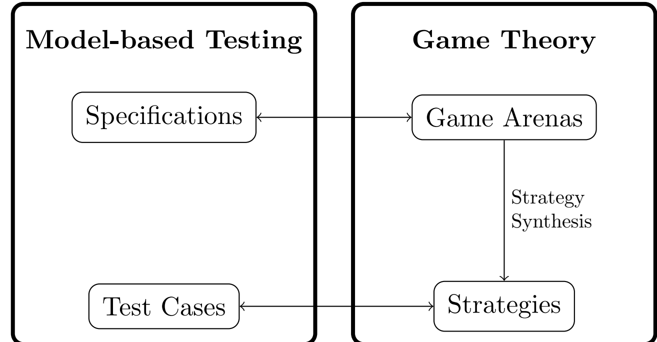

Fig. 1. Model-based testing via game theory overview

In their framework [1], Van den Bos & Stoelinga identify and formalise four distinct test assumptions, and note a shared commonality in that they always ignore either an input or an output action. It is conjectured that a test assumption which takes into account both actions by executing both but in nondeterministic order could prove to be beneficial for the modeling and testing of concurrent software systems.

In this work, the conjectured concurrent test assumption is formalised and the framework found in [1] is extended with necessary novel construct to deal with concurrent systems. Its efficacy is discussed and multiple avenues for future research are identified.

2 PRELIMINARIES

The framework on which the current research expands, was put forward by Van den Bos & Stoelinga in [1]. In the following section, a recapitulation of the relevant sections of this work will be given, as they contain foundational definitions and constructs for our work.

All definitions in Section 2 are taken from [1] (modulo notation).

2.1 Games

In Section 2.3, elements from model-based testing will be likened to analogous elements from game theory. This section defines required preliminary game theoretical constructs.

2.1.1 Game arenas. We consider concurrent two-player games, played on graphs called game arenas.

Definition 2.1. A *game arena* is a tuple $G = (Q, q_0, Act, Act_2, \Gamma_1, \Gamma_2, Moves)$ 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,

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
TScIT 41, July 5, 2024, Enschede, The Netherlands
© 2024 University of Twente, Faculty of Electrical Engineering, Mathematics and Computer Science.

- $\Gamma_i : Q \to 2^{Act_i} \setminus \emptyset$ 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 Players 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 $a \notin \Gamma_1(q) \vee x \notin \Gamma_2(q)$.

2.1.2 *Plays.*

Plays are the infinite sequences of states and actions obtained after tracking all events in a game. They are finitely described by their prefixes.

Definition 2.2. A play π 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 $\forall j \in \mathbb{N} : a_j \in \Gamma_1(q_j), x_j \in \Gamma_2(q_j)$.

We write:

$$\pi_j^q \stackrel{\text{def}}{=} q_j \qquad \text{for the j-th state,}$$

$$\pi_j^a \stackrel{\text{def}}{=} a_j \qquad \text{for the Player 1 action, and}$$

$$\pi_j^x \stackrel{\text{def}}{=} x_j \qquad \text{for the Player 2 action.}$$

Furthermore, 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 $\Pi^{pref}(G) \stackrel{\text{def}}{=} Pref(\Pi(G))$.

A play is winning if it passes through some state in a reachability goal $R \subseteq Q$.

Definition 2.3. 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. We write $Win\Pi(G, R)$ for the set of winning plays with respect to R.

Formally:

$$Win\Pi(G, R) = \left\{\pi \in \Pi(G) \mid \exists j \in \mathbb{N} : \pi_j^q \in R\right\}$$

2.2 Model-based testing

Model-based testing facilitates the efficient testing of software systems by allowing the tester to automate the generation of test cases. This is achieved by creating a specification model of the system under test (SUT), which is subsequently used to generate test cases through a variety of techniques [3, 4, 7]. Specifications describe the desired behaviour of the system and are given in the form of automata with inputs and outputs.

2.2.1 *System specifications as suspension automata.*

We use suspension automata (SAs) to model system specifications.

We first define some notation for partial functions.

Definition 2.4. Let $f : X \rightharpoonup Y$ be a partial function. We adopt the following notations:

- $f(x)\downarrow$ if $f(x)$ is defined, and
- $f(x)\uparrow$ if $f(x)$ is undefined.

Definition 2.5. 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,
- $L_O^\delta = L_O \bigcup\{\delta\}$ with L_O a finite set of output labels, $\delta \notin L_O$ and $L_I \bigcap L_O = \emptyset$,
- $T : Q \times (L_I \bigcup L_O^\delta) \rightharpoonup Q$ is a partial transition function, and,
- $q_0 \in Q$ is an initial state.

We additionally require that an SA is non-blocking, i.e.:

$$\forall q \in Q : out(q) \neq \emptyset$$

Lastly, we define:

- $L \stackrel{\text{def}}{=} L_I \bigcup L_O^\delta$,
- The enabled inputs and outputs in a state $q \in Q$ are respectively

$$in(q) = \{a \in L_I \mid T(q, a) \downarrow\}, \text{ and}$$

$$out(q) = \{x \in L_O^\delta \mid T(q, x) \downarrow\}.$$

The label δ is used to indicate quiescence, i.e., the absence of an observable output.

2.2.2 *Test assumptions.*

It is possible for a single state to have both input transitions and output transitions defined from it. Such states are called *mixed states.* If, in such a state, the tester provides an input at the same time that the SUT provides an output, an *input-output conflict* arises. This conflict is solved with the use of a *test assumption*, which prescribes to which state(s) the system could transition. Note the possible plural here; a test assumption may define multiple successive states after an input-output conflict and may therefore introduce a degree of non-determinism, which is resolved upon test case execution.

In [1], Van den Bos & Stoelinga identify four test assumptions. For the sake of future comparison, they are reiterated here:

- *input-eager (IE)*: The tester is always able to provide an input. In the case of a conflict, the output will be discarded.
- *output-eager (OE)*: The converse of *IE*; the SUT is always able to produce an output. In the case of a conflict, the input will be discarded.
- *non-deterministic (ND)*: It is determined non-deterministically whether the SUT will be able to produce an output, or the tester will be able to produce in input in the case of a conflict.
- *input-fair (IF)*: The tester is guaranteed to be able to provide an input after a finite number of attempts.

2.2.3 *Test cases.*

Test cases are finite acyclic SAs that prescribe which actions the tester is to take to test one or more elements of the SUT and whose transitions eventually lead to *Pass* or *Fail* verdicts.

We first define some auxiliary notation.

Definition 2.6. 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:

$$q \textit{ after } \epsilon = \{q\}$$
$$q \textit{ after } \mu\rho = \begin{cases} T(q,\mu) \textit{ after } \rho & \text{if } T(q,\mu)\downarrow \\ \emptyset & \text{otherwise} \end{cases}$$
$$\mathscr{A} \textit{ after } \rho = q_0 \textit{ after } \rho$$
$$out(Q') = \bigcup_{q' \in Q'} out(q')$$
$$traces(\mathscr{A}) = \{\rho' \in L^* \mid \mathscr{A} \textit{ after } \rho' \neq \emptyset\}$$

From a specification $\mathscr{A}$, we construct a test case $\mathscr{T}$ by repeatedly taking one of the following steps:

(1) Choose an input $a? \in in(q)$ (where $q \in Q$ is the current state in $\mathscr{A}$) and execute it, moving to the next state in $\mathscr{T}$. Enable all outputs from L_O.
(2) Observe an output from the SUT. If the observed output is not permitted by the specification, emit a *Fail* verdict, otherwise move to the next state in $\mathscr{T}$.
(3) Stop testing and emit a *Pass* verdict.

Definition 2.7. 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
  $\forall x \in L_O^\delta : T^t(Pass, x) = Pass \land T^t(Fail, x) = Fail$, and
  $\forall a \in L_I : T^t(Pass, a) \uparrow$.
- $\mathscr{T}$ has no cycles except those in *Pass* and *Fail*.
- Every state enables all outputs L_O, and either one input or δ:

$$\forall q \in Q : (|in(q)| = 0 \land out(q) = L_O^\delta) \lor (out(q) = L_O \land |in(q)| = 1)$$

  Exception: in case of an input-eager test interaction, we only require:

$$\forall q \in Q : (|in(q)| = 0 \land out(q) = L_O^\delta) \lor |in(q)| = 1$$

- Traces of $\mathscr{T}$ leading to *Pass*, are traces of $\mathscr{A}$, while traces to *Fail* are not:

$$\forall \rho \in traces(\mathscr{T}) : (\mathscr{T} \textit{ after } \rho = Pass \implies \rho \in traces(\mathscr{A})) \land (\mathscr{T} \textit{ after } \rho = Fail \implies \rho \notin traces(\mathscr{A}))$$

2.3 Specifications are game arenas

As discussed in [1], the model-based testing of software systems closely resembles the structure of concurrent two-player games. Each turn the two players (the tester and the SUT) propose actions (inputs and outputs); based on whose predefined interactions (the test assumptions) the system moves to a new state.

Since testers analyse their interactions with the SUT via the traces of test cases, it is important to track for each state which player it was that executed the action to end up in this state. To achieve this, the state space is extended with a number $i \in \{1, 2\}$, indicating the state was reached by Player i. The Player 1 actions are extended with θ and $stop?$, meaning respectively that the tester does not want to provide an input and that the tester wants to stop testing. The state space is additionally extended with a sink state $\bot$, which is reached after the $stop?$ action.

Definition 2.8. Let $\mathscr{A} = (Q, L_I, L_O^\delta, T, q_0)$ be an SA. The *game arena underlying* $\mathscr{A}$ is defined by
$G_\mathscr{A} = (Q_\bot, (q_0, 1), Act_1, Act_2, \Gamma_1, \Gamma_2, Moves)$ where:

- $Q_\bot = (Q \times \{1, 2\}) \cup \{(\bot, 1)\}$,
- $Act_1 = L_1 \cup \{\theta, stop?\}$ and $Act_2 = L_0^\delta$,
- for all $q \in Q$ and $i \in \{1, 2\}$, we take
  $\Gamma_1((q, i)) = in(q) \cup \{\theta, stop?\}$ and
  $\Gamma_2((q, i)) = out(q)$,
- we take
  $\Gamma_1((\bot, 1)) = \{stop?\}$ and
  $\Gamma_2((\bot, 1)) = L_O^\delta$, and
- the function $Moves : Q_\bot \times Act_1 \times Act_2 \to 2^{Q_\bot}$ encodes one of the different test assumptions.

Besides the requirement from Definition 2.1 for moves with undefined action, we require
$Moves(q, stop?, x) = \{(\bot, 1)\}$, and
$Moves((\bot, 1), stop?, x) = \{(\bot, 1)\}$.

In the subsequent, we fix a specification $\mathscr{A}$ and its underlying game $G_\mathscr{A}$.

2.3.1 *Encoding test assumptions.*

Different test interactions are formalised by implementing a *Moves* function for each test assumption. The *Moves* function takes a game state $q \in Q$, an input action $a \in Act_1$ and an output action $x \in Act_2$ and gives a set of states to which the system could move next.

For the sake of future comparison we give the definitions of the *Moves* functions for the input-eager and non-deterministic test assumptions. $Moves_{IE}$ always executes the input action, unless the input is θ, indicating the user waits for the system to produce an output, whereas $Moves_{ND}$ leaves it undetermined which action is executed except if the tester decides to nothing, in which case the output is guaranteed to be executed, and vice versa.

Definition 2.9.

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

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

3 CONCURRENT INTERACTIONS

A test interaction is seen as a system's behaviour following the simultaneous occurrence of an input and an output. Some systems are designed in such a way that the user will always be able to provide an input; think of a music player, which always responds to a user's input to stop playing, in stead of ignoring the input in favour of outputting music forever. For such a system, an input-eager test assumption may correctly model the desired system behaviour. For other systems, such as a vending machine, it is crucial that the output be always fully executed before a new input is processed; therefore an output-eager test assumption may be better suited. After all, a customer would be quite disappointed if the candy bar they just bought never got dispensed, because they accidentally pushed a button.

In reality however, many systems are more complex, and their behaviour after an input-output conflict cannot be accurately modelled by simply discarding one or the other; often, it is influenced by both. We display one possible such system in Figure 2 and discuss its behaviour in Example 3.1.

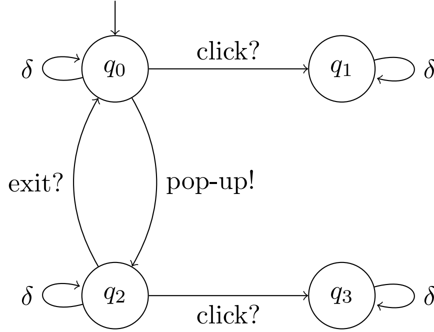

Fig. 2. SA of a website with pop-up

Example 3.1. In Figure 2, we show a minimal SA of an imaginary website, with a single button for the user to click, leading to some different page (transitioning from q_0 to q_1). The website has the frustrating property however, that while in q_0, at any moment a pop-up advertisement may appear, covering the button (represented by state q_2), which when clicked leads to a different page (q_3). The pop-up can be exited by clicking an exit button (leading the user back to q_0).

In state q_0, if the user clicks the button (gives the input *click?*) at the same time that the system displays the pop-up (gives the output *pop-up!*), one of two things can happen:

(1) The button get clicked, and the system transitions to q_1.
(2) The advertisement gets clicked, and the system transitions directly to q_3.

This behaviour cannot be correctly modelled with one of the test assumptions from Section 2.2.2, since in the case of a conflict in q_0 they will all transition to either q_1 or q_2; the latter being disallowed in our (verbal) description of the system. Based on this observation, we gather that a new (concurrent) test assumption is required to model this system.

3.1 Concurrency in SAs

We evaluate the ability of Suspension Automata, as defined in Definition 2.5, to handle concurrent interactions. The transition function T of an SA takes a state and a single action, and returns the next state to which the system will transition. To model concurrent interactions, we need to be able to transition between states based on both an input and an output. Therefore, we must slightly alter our definition of an SA.

3.1.1 Double-step transitions. We first extend an SA's transition function T with the notion of a *double-step transition*:

Definition 3.2. Let $\mathscr{A} = (Q, L_I, L_O^\delta, T, q_0)$ be an SA, $q \in Q$ and $\mu, \nu \in L$, such that: $\mu \in L_I \Leftrightarrow \nu \in L_O$ (i.e., one is an input, the other an output). Then we define a *double-step transition* function $T^2 : Q \times (L_I \bigcup L_O^\delta)^* \rightharpoonup Q$ as follows:
For sequences of length 1, T^2 behaves identically to T:

$$T^2(q, \mu) = T(q, \mu)$$

For sequences of length 2, the following holds:

$$\text{if } \; T(q,\mu)\downarrow \wedge\, T(T(q,\mu),\nu)\downarrow : T^2(q,\mu\nu) = T(T(q,\mu),\nu)$$
$$\text{otherwise } \; : T^2(q,\mu\nu)\uparrow$$

In any other case the transition is undefined.

3.1.2 Concurrent SAs. We then add support for double-step transitions to SAs by defining a *concurrent SA* as such:

Definition 3.3. A *concurrent suspension automaton* (SA_C) is an SA $\mathscr{A}_C = (Q, L_I, L_O^\delta, T^2, q_0)$ as defined in Definition 2.5, with the following alteration:

- $T^2 : Q \times (L_I \bigcup L_O^\delta)^* \rightharpoonup Q$ is the partial double-step transition function (as defined in Definition 3.2).

3.2 Concurrency in games

We evaluate the ability of game arenas, as defined in Definition 2.1, to handle concurrent interactions. A *Moves* function of a game arena takes a state, an input action and an output action and returns the next state to which the game will transition. So it is possible to define a transition based on both the input and the output, and there is therefore no need to make any alterations to its definition.

3.3 From concurrent SAs to game arenas

A concurrent SA may be translated to a game arena in much the same way as defined in Definition 2.8, with some slight alterations.

In Definition 2.8, the game's state space $Q_\perp$ was extended with the annotations 1 and 2 to record which action was executed, such that the tester may see from the trace what occurred during test case execution. To preserve this property with double-step transitions, we additionally extend the state space with the annotations 12 and 21, indicating a Player 1 transition followed by a Player 2 transition and vice versa respectively. We then define the game arena underlying a concurrent SA as:

Definition 3.4. Let $\mathscr{A}_C = (Q, L_I, L_O^\delta, T^2, q_0)$ be a concurrent SA. The *game arena underlying* $\mathscr{A}_C$ is defined identically to Definition 2.8 as $G_{\mathscr{A}_C} = (Q_\perp, (q_0, 1), Act_1, Act_2, \Gamma_1, \Gamma_2, Moves)$, with the following exceptions:

- $Q_\perp = (Q \times \{1, 2, 12, 21\}) \cup \{(\perp, 1)\}$
- for all $q \in Q$ and $i \in \{1, 2, 12, 21\}$, we take
  $\Gamma_1((q, i)) = in(q) \cup \{\theta, stop?\}$ and
  $\Gamma_2((q, i)) = out(q)$.

3.4 The concurrent test assumption

Given some current state q_0, an input action $a?$ and an output action $x!$ and the possibility of actions occurring concurrently (in the form of double-step transitions); we discern the six possible different situations of q_0 and its surrounding states found in Figure 3:

(1) q_0 only has an output $x!$ enabled.

$$in(q_0) = \emptyset \wedge x! \in out(q_0)$$

(2) q_0 has a quiescent output δ and an input $a?$ enabled.

$$a? \in in(q_0) \land out(q_0) = \delta$$

(3) q_0 has an input $a?$ and an output $x!$ enabled. Additionally, the state reached by $a?$ has $x!$ enabled and the state reached by $x!$ has $a?$ enabled.

$$a? \in in(q_0) \land x! \in out(q_0) \land x! \in out(T(q_0, a?)) \land a? \in in(T(q_0, x!))$$

(4) q_0 has an input $a?$ and an output $x!$ enabled. Additionally, the state reached by $a?$ has $x!$ enabled but the state reached by $x!$ does not have $a?$ enabled.

$$a? \in in(q_0) \land x! \in out(q_0) \land x! \in out(T(q_0, a?)) \land a? \notin in(T(q_0, x!))$$

(5) q_0 has an input $a?$ and an output $x!$ enabled. Additionally, the state reached by $a?$ does not have $x!$ enabled but the state reached by $x!$ does have $a?$ enabled.

$$a? \in in(q_0) \land x! \in out(q_0) \land x! \notin out(T(q_0, a?)) \land a? \in in(T(q_0, x!))$$

(6) q_0 has an input $a?$ and an output $x!$ enabled. Additionally, the state reached by $a?$ does not have $x!$ enabled and the state reached by $x!$ also does not have $a?$ enabled.

$$a? \in in(q_0) \land x! \in out(q_0) \land x! \notin out(T(q_0, a?)) \land a? \notin in(T(q_0, x!))$$

A test interaction in a concurrent SA is considered *concurrent (C)* if it is determined non-deterministically to which subsequent state the system will transition, using either a normal "single-step" transition or, if possible, a double-step transition. That is, if for each of the six possible situations in Figure 3, it is determined non-deterministically to which of the states highlighted in blue the system will transition.

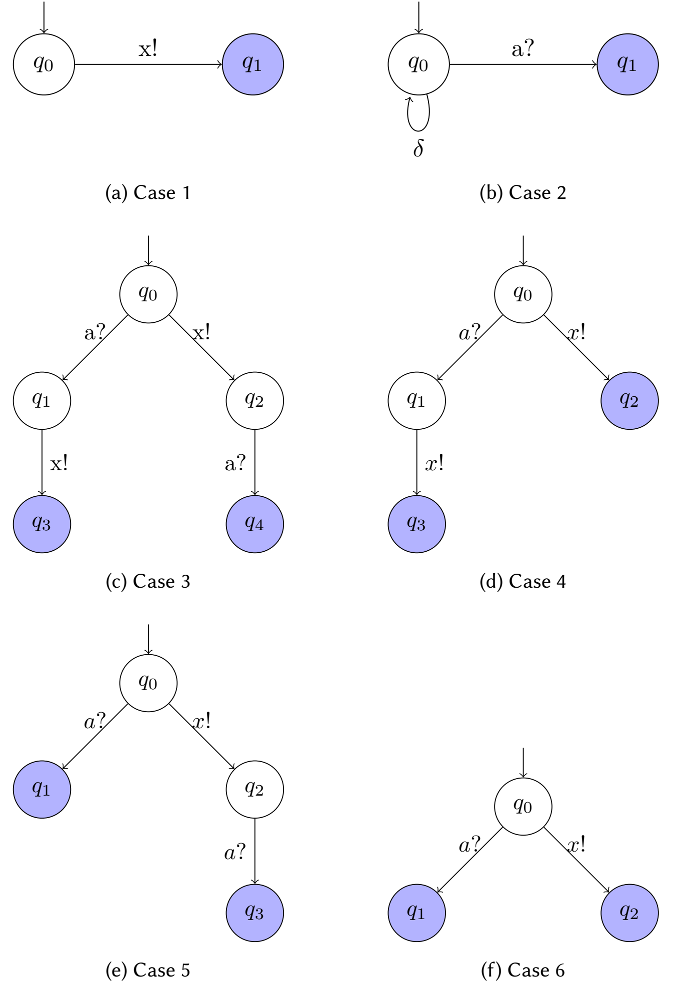

Fig. 3. Concurrent cases

3.5 Encoding the concurrent test assumption

To formalise this concurrent test assumption and use it on a specification's underlying game arena $G_{\mathscr{A}_C}$, we define a new implementation of $G_{\mathscr{A}_C}$'s *Moves* function in the spirit of Definition 2.9.

Definition 3.5. Let $\mathscr{A}_C = (Q, L_I, L_O^{\delta}, T^2, q_0)$ be a concurrent *SA*. The function $Moves_C : Q_\perp \times (L_I \bigcup \{\theta, stop?\}) \times L_O^{\delta} \to 2^{Q_\perp}$ encodes the concurrent test assumption described in Section 3.4:

$$Moves_C((q,i),a,x) = \begin{cases} \{(T^2(q,x),2)\} & \text{if } a = \theta \\ \{(T^2(q,a),1)\} & \text{if } x = \delta \\ \{(T^2(q,xa),21),(T^2(q,ax),12)\} & \text{if } a \in in(T^2(q,x)) \\ & \quad \land x \in out(T^2(q,a)) \\ \{(T^2(q,ax),12),(T^2(q,x),2)\} & \text{if } x \in out(T^2(q,a)) \\ \{(T^2(q,xa),21),(T^2(q,a),1)\} & \text{if } a \in in(T^2(q,x)) \\ \{(T^2(q,a),1),(T^2(q,x),2)\} & \text{otherwise} \end{cases}$$

Note that the six cases of the $Moves_C$ function definition correspond exactly with the six cases in Figure 3. We briefly explain each case (assuming that all proposed actions are enabled in the relevant states):

(1) If $a = \theta$ (i.e., the tester provides no input): execute $x!$.
(2) If $x = \delta$ (i.e., the SUT provides no output): execute $a?$.
(3) If it would be possible to take both the double-step $a?x!$ and $x!a?$, take either.
(4) If it would be possible to take the double-step $a?x!$ but not $x!a?$, execute either $a?x!$ or $x!$.
(5) If it would be possible to take the double-step $x!a?$ but not $a?x!$, execute either $x!a?$ or $a?$.
(6) If it is not possible to take any double-step transitions, take either $a?$ or $x!$ (identical to the non-deterministic test assumption).

Note that the resulting states in cases 3 to 6 (corresponding to the highlighted states in Figures 3c to 3f) are not necessarily distinct.

3.6 Concurrent test cases

Test cases must be altered slightly in order to take into account possible double-step transitions. We therefore define a *concurrent test case* as follows:

Definition 3.6. A *concurrent test case* for an SA $\mathscr{A}$ is an SA $\mathscr{T}_C = (Q^t, L_I, L_O^\delta, T^t, q_0^t)$ as defined in Definition 2.7, with the following alterations:

- The transition function T^t is a double-step transition function from Definition 3.2.
- For every mixed state in the specification, if a transition is added to the test case which in the specification could be a double-step transition, also add the second action from the double-step transition.

4 APPLICABILITY

We evaluate the applicability and usefulness of the newly defined concurrent test assumption, once again using the system specification depicted in Figure 2 as an example.

4.1 Modelability

In this section we consider the 'modelability' of systems using a concurrent test assumption as opposed to using existing assumptions (Section 2.2.2). Under modelability we understand the degree to which a system is able to be modeled, using the given tools.

Consider the following two play prefixes, using a non-deterministic and concurrent test assumption respectively, through the game arena $G_{\mathscr{A}_C}$ underlying the concurrent SA $\mathscr{A}$ in Figure 2:

$$\pi_{ND} = (q_0, 1)\,\langle \textit{click?}, \textit{pop-up!}\rangle\,(q_2, 2)\,\langle \textit{exit?}, \delta\rangle\,(q_0, 1)$$
$$\pi_C = (q_0, 1)\,\langle \textit{click?}, \textit{pop-up!}\rangle\,(q_3, 21)$$

Recall that in Example 3.1 we defined the real-life behaviour of this system as directly transitioning to q_3 if in q_0 the user clicks the button at the same time that the website displays the pop-up. In π_{ND} we see in the first turn, the users click is ignored, allowing the user to subsequently close the pop-up with an *exit*? action. This is not how the system should behave, therefore this particular system cannot be accurately modeled with a non-deterministic test assumption. The same would go for the input-eager, output-eager and input-fair assumptions; they all transition to either q_1 and/or q_2.

In π_C the user clicks the button at the same time that the website displays the pop-up, and the system transitions directly to q_3, simultaneously executing both the input and the output. This is the desired behaviour of the system; therefore the concurrent test assumption seems to allow us to correctly model a system which could not be modeled by existing assumptions.

4.2 Resource efficiency

Due to its ability to take double-step transitions, test execution under a concurrent test assumption may require less time and resources than under its existing counterparts. Consider for example the two following play prefixes of $G_{\mathscr{A}_C}$:

$$\pi_{ND} = (q_0, 1)\,\langle \textit{click?}, \textit{pop-up!}\rangle\,(q_2, 2)\,\langle \textit{click?}, \delta\rangle\,(q_3, 1)$$
$$\pi_C = (q_0, 1)\,\langle \textit{click?}, \textit{pop-up!}\rangle\,(q_3, 21)$$

Both prefixes end in q_3, however π_C requires one fewer turn to arrive there. From this fact, we can see that the concurrent test assumption might provide an advantage in the form of reduced runtimes during test execution. This attribute however, is offset by the fact that test case generation under the concurrent test assumption requires more time and memory due to the look-ahead imposed by double-step transitions. While with e.g. an input-eager assumption a number of unreachable states may be omitted from a test case, a concurrent assumption requires not only all outputs to be enabled but also subsequent inputs if their inclusion allows for a double-step transition.

Therefore the degree to which the concurrent test assumption provides an increase or decrease in total resource efficiency relies on how often a test case is reused; if a test case gets generated once and is subsequently used a hundred times, then the time lost in generation may well be made up for during repeated execution. Of course, the opposite also holds true; if a test case is executed only once after generation, the total efficiency is likely to be worse with a concurrent assumption than it would have been with another.

5 DISCUSSION

In [1], Van den Bos & Stoelinga conjecture that a concurrent interpretation of input-output conflicts may prove to be beneficial or well-suited for systems dealing with concurrent processes. After formalising this concurrent test interaction, we have demonstrated that it may indeed proof to be beneficial when working with concurrent software systems, as it allows the accurate modeling of concurrent interactions, which was not possible with existing test assumptions (as demonstrated in Section 4.1). Some findings in this research however, are far from conclusive, and show no concrete results in the form of real test generation and execution using the new concurrent test assumption. We will discuss the results and shortcomings of this research and provide recommended avenues of approach for future work.

5.1 Shortcomings

As it was considered out of scope given the limited temporal constraints imposed on this research, the proposed framework does not provide formal definitions or descriptions on the relation between concurrent test cases and game strategies (discussed in [1]). Additionally the *ioco conformance relation* [8, 6] and its correlation to *alternating trace inclusion* (also discussed in [1]) remain unexplored in the context of a concurrent test assumption.

Furthermore, a concrete software implementation of the concurrent test assumption is lacking. Such an implementation is a crucial next step for this line of research, as it would allow us to draw more significant conclusions based on empirical evidence.

5.2 Future work

As mentioned in Section 5.1, a formal description of the relation between concurrent test cases and game strategies is lacking from the current paper. The most pressing matter for future work would be to thoroughly investigate this relation and the implications a concurrent test case may have for test case generation via game strategies.

Secondly, as discussed in Section 5.1, a software implementation of a model-based testing application (making use of game strategy synthesis techniques) would enable the collection of empirical data,

which may be used to draw stronger conclusions on the area of resource efficiency, as discussed in Section 4.2, in addition to verifying whether the proposed framework holds water.

Furthermore, the effects of a concurrent test assumption on the ioco conformance relation remain unexplored. It might be intriguing to see if the concurrent test assumption has any interesting implications for the ioco relation, and whether it proves beneficial there.

Lastly, in this framework it was decided to implement concurrency by subsequent execution of actions in the span of one, while leaving states and transitions unchanged. Perhaps a different approach, such as replacing double-step transitions with a single action in which both occur truly simultaneously, may provide useful results.

6 CONCLUSION

We have formalised the concurrent test assumption conjectured by Van den Bos & Stoelinga in [1] and its efficacy has been analysed. Based on our evaluations, we conclude that the novel test assumption may allow for the accurate modeling of certain systems dealing with concurrent processes, in addition to possibly providing an increase in overall testing efficiency. Empirical validation is still required to evaluate real world effectiveness. We have identified multiple avenues for further academic study.

ACKNOWLEDGMENTS

I would like to thank dr. Petra van den Bos for her help in understanding the source material, her useful feedback and supervision over the duration of this research.

Furthermore, I would like to thank my wonderful girlfriend Bente for her continued aid and support over the last months.

REFERENCES

[1] 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. arXiv:1809.03098 [cs]. DOI: 10.4204/EPTCS.277.9.

[2] 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, (Mar. 2008), 486–491. ISBN: 978-3-9810801-3-1. DOI: 10.1145/1403375.1403491.

[3] A. Hartman and K. Nagin. 2004. The AGEDIS tools for model based testing. en. *Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis - ISSTA '04*. DOI: 10.1145/1007512.1007529.

[4] Claude Jard and Thierry Jéron. 2005. TGV: theory, principles and algorithms. en. *International Journal on Software Tools for Technology Transfer*, 7, 4, (Aug. 2005), 297–315. Company: Springer Distributor: Springer Institution: Springer Label: Springer Number: 4 Publisher: Springer Berlin Heidelberg. DOI: 10.1007/s10009-004-0153-x.

[5] 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. DOI: 10.1145/1013886.1007520.

[6] Mark Timmer, Hendrik Brinksma, and Mariëlle Ida Antoinette Stoelinga. 2011. Model-Based Testing. Undefined. In *Software and Systems Safety: Specification and Verification*. IOS, (Apr. 2011), 1–32. DOI: 10.3233/978-1-60750-711-6-1.

[7] G. J. Tretmans and Hendrik Brinksma. 2003. TorX: Automated Model-Based Testing. Undefined. In *First European Conference on Model-Driven Software Engineering*. (Dec. 2003), 31–43. Retrieved May 3, 2024 from https://research.utwente.nl/en/publications/torx-automated-model-based-testing.

[8] Jan Tretmans. 2008. Model Based Testing with Labelled Transition Systems. en. 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. ISBN: 978-3-540-78917-8. DOI: 10.1007/978-3-540-78917-8_1.

[9] Farn Wang, Sven Schewe, and Jung-Hsuan Wu. 2015. Complexity of node coverage games. *Theoretical Computer Science*, 576, (Apr. 2015), 45–60. DOI: 10.1016/j.tcs.2015.02.002.