Symbolic Input-Output Conformance Checking for Model-Based Mutation Testing

Bernhard K. Aichernig and Martin Tappler[1]

Institute for Software Technology
Graz University of Technology, Austria

Abstract

This paper presents an approach to use symbolic input output conformance checking for mutation-based test case generation. In this approach, a possibly non-deterministic action system model is used as basis for generating a number of mutants. Subsequently after the generation of mutants, the original model and the mutants are simultaneously symbolically executed and tested for conformance. Distinguishing test cases are generated, if non-conformance is detected during this process. Several optimisations of the conformance check are presented and their effectiveness is underpinned by listing experimental results.

Keywords: model-based testing, mutation testing, symbolic execution, action systems, sioco,ioco.

1 Introduction

In order to assure that a system under test (SUT) fulfils given requirements it is commonly executed and tested under conditions specified through a set of test cases. Traditional software testing however suffers from a number of drawbacks. It is for instance inherently incomplete and since it is a manual task, it is labour intensive and error-prone. Model-Based Testing aims at improving upon this situation, by utilising abstract models of the SUT [21]. Most importantly, models allow the automatic generation of test cases based on some criterion. Hence, the ad hoc nature of software testing is replaced by a well-defined process.

Model-Based Mutation Testing uses a fault-based approach to test case selection [6]. More concretely, mutation-based test case generation consists of two main steps: (1) mutated models are generated by injecting faults into the original model and (2) test cases are generated, which would reveal non-conforming behaviour of mutants. The rationale behind this approach is that a SUT implementing a non-conforming mutant would be detected to be faulty by the test case corresponding to the mutant.

There is a variety of conformance relations in use today, like refinement [5] or Input Output Conformance (**ioco**) [2]. In the following, we will use Symbolic

[1] Authors are listed in alphabetical order.

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: `www.elsevier.nl/locate/entcs`

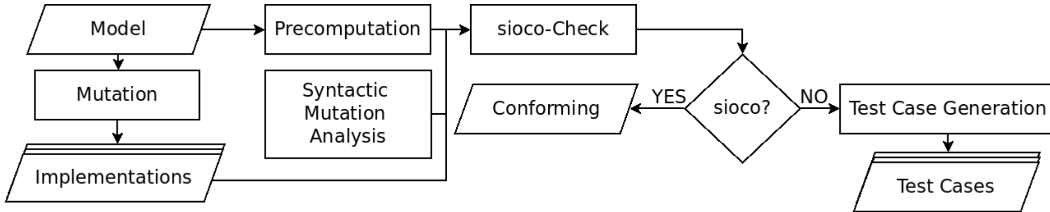

Fig. 1. The model-based mutation test case generation process.

Input Output Conformance (**sioco**) as defined by Frantzen et al. [14] to decide conformance between mutants and the original model. We have chosen **sioco** as it is well-suited for reactive systems and its symbolic nature provides a solution to the state space explosion faced when working concretely. However, we rather test for non-conformance than for conformance in the test case generation process depicted in Figure 1. More specifically, we generate sequences of actions and conditions leading to states where non-conformance of first-order mutants, i.e. mutants created by injecting a single fault, may be observed.

The contribution of our work is twofold. First, we give a formalisation of the subsequently introduced variant of the action system formalism. For this purpose, we will adapt the symbolic framework given in [14] and follow the same style. Second, we will present an efficient fully symbolic check for non-conformance between two action systems, which may behave non-deterministically. We demonstrate the efficiency through a comparison with a previously implemented concrete **ioco** checker.

2 Action Systems

Action systems were first defined by Back and Kurkio-Suonio [7] as a modelling formalism for distributed systems. We have chosen this formalism as it can effectively be used for modelling reactive systems [8] and because recently, it has also been used for model-based mutation testing [5]. There exist several variations of it like object-oriented action systems [9] and it also served as an inspiration for Event-B [1]. However, the action system formalism used here is more restricted than other variations. In some aspects it is similar to the Event-B language, but for instance it does not support set-theoretic constructs like Event-B.

Informally, the execution of an action system starts in an initial state which is manipulated by repeatedly executing actions. During this process one action is chosen at each step in a non-deterministic fashion from the set of enabled actions. An action is enabled iff its guard is satisfiable in the current state. The execution terminates when the set of enabled actions is empty.

2.1 Syntax

A definition of the syntax of action systems is given in Figure 2, in which overlines denote possibly empty repetitions of elements and bold-faced strings denote terminal symbols as in the syntax definition given in [5]. An action system definition starts with the definition of a name given as a capitalised identifier and contains the definition of types, the declaration of state variables, the initial state and actions.

$$
\begin{array}{rl}
AS ::= & \texttt{def}\ capid\ \texttt{\{}T\ S\ I\ ACT\texttt{\}} \\
T ::= & \texttt{types}\ \texttt{\{}\overline{capid\texttt{=}TYPE}\texttt{\}} \\
TYPE ::= & \texttt{[}int\texttt{..}int\texttt{]} \mid \texttt{[}\overline{capid}\texttt{]}capid\texttt{]} \\
S ::= & \texttt{state}\ \texttt{\{}\overline{id\texttt{:}ty}\texttt{\}} \\
I ::= & \texttt{init}\ B \\
ACT ::= & \texttt{actions}\ \texttt{\{}\overline{A}\texttt{\}} \\
A ::= & (\texttt{?} \mid \texttt{!} \mid \epsilon)\ id(\overline{id\texttt{:}ty})\ \texttt{if}\ E\ \texttt{then}\ B
\end{array}
\qquad
\begin{array}{rl}
B ::= & \texttt{\{}\overline{id\texttt{:=}\ E}\texttt{\}} \\
E ::= & id \mid C \mid E\texttt{+}E \mid \texttt{!}E \mid \texttt{(}E\texttt{)} \\
& \mid E\texttt{==}E \mid E\texttt{<}E \mid \ \ldots \\
C ::= & \texttt{True} \mid \texttt{False} \mid capid \mid int \\
id ::= & \text{identifier} \\
capid ::= & \text{capitalised identifier} \\
int ::= & \text{integer number literal}
\end{array}
$$

Fig. 2. The action system syntax.

The types block consists of a list of type definitions, which associate type names with user-defined types. The init block contains one assignment with an expression over constant terms as right-hand side for each state variable. This block is followed by the actions block which defines an arbitrary number of actions. Every action definition consists of a label definition, a parameter list, a boolean expression called guard and at most one assignment per state variable. The label definition optionally starts with a question or an exclamation mark, followed by an identifier, which defines the name of the action. A question mark denotes the action as input, an exclamation mark as output and the absence of both denotes it as internal action.

2.2 Semantics

The semantics of action systems is usually defined using weakest precondition formulae [10], but in the following a semantics, which relates action systems to Input Output Labelled Transition Systems (IOLTSs) will be given. The semantics and definitions will closely follow the definitions of Symbolic Transition Systems (STSs) [14] . This approach was taken, because action systems can easily be translated to initialised Input Output Symbolic Transition Systems (IOSTSs). This way it is possible to use the symbolic framework given in [14] with some adaptations. Moreover, we will use the same concepts and notation for first-order formulae as follows.

Conventions

Generally, we assume the usage of one-sorted logic with a non-empty set of possible values called universe and denoted by $\mathfrak{U}$. We will denote the set of terms containing variables from a set X by $\mathfrak{T}(X)$ and first-order formulae containing free variables from the same set by $\mathfrak{F}(X)$. The set of all total functions from A to B shall be denoted by B^A, thus a valuation for variables in X is given by $\mathfrak{U}^X$. The union of two valuations $\upsilon \in \mathfrak{U}^X$ and $\varsigma \in \mathfrak{U}^Y$ is a valuation for variables in $X \cup Y$ and shall be denoted $\upsilon \cup \varsigma$. To denote the evaluation of terms based on a valuation υ we use the notation υ_{eval}, i.e. υ_{eval} is a function mapping terms to the universe $\mathfrak{U}$, and we use the function $eval : \mathfrak{T}(\emptyset) \to \mathfrak{U}$ to evaluate constant terms. The substitution of variables shall be denoted by $g[\sigma]$, where σ is a function from variables to terms and g is some formula or term. A short-hand notation for existential quantification over variables from a set X will be used, $\overline{\exists}_X p$ shall denote $\exists x_1, \ldots \exists x_n : p$ for $x_1, \ldots x_n \in X$. Finally, f_X will be used to restrict the domain of a function f to X.

Definition 2.1 [Abstract Syntax of Action Systems] An action system is a tuple $\mathcal{AS} = \langle \mathcal{V}, \mathcal{I}, \Lambda_I, \Lambda_U, \iota, \rightarrow \rangle$, where $\mathcal{V}$ is the set of state variables and $\mathcal{I}$ is the set of

parameter variables [2], with $\mathcal{V} \cap \mathcal{I} = \emptyset$ and $Var = \mathcal{V} \cup \mathcal{I}$. $\Lambda = \Lambda_I \cup \Lambda_U$ is the set of action labels, with Λ_I being the set of input actions and Λ_U being the set of output actions. The constant $\tau \notin \Lambda$ denotes an internal action and we set $\Lambda_\tau = \Lambda \cup \{\tau\}$. The initialisation of the action system is $\iota \in \mathfrak{T}(\emptyset)^{\mathcal{V}}$. The set $\rightarrow \subseteq \Lambda_\tau \times \mathfrak{F}(Var) \times \mathfrak{T}(Var)^{\mathcal{V}}$ is the transition relation. For $(\lambda, \varphi, \rho) \in \rightarrow$, λ is called label, φ is called guard, ρ is the update mapping, defined by assignments in the action body.

Similarly to [14], the following functions and vocabulary will also be used:

(i) $arity : \Lambda_\tau \rightarrow \mathbb{N}_0$ associates each action with its number of parameters

(ii) The function $para$ associates each action λ with a tuple of size $arity(\lambda)$ containing the parameter variables for λ.

(iii) For all actions λ, $para$ maps λ to a tuple of distinct parameter variables and for $(\lambda, \varphi, \rho) \in \rightarrow$ it holds that $free(\varphi) \subseteq \mathcal{V} \cup para(\lambda)$ and $\rho \in \mathfrak{T}(\mathcal{V} \cup para(\lambda))^{\mathcal{V}}$.

As in [14], we only consider well-defined models. An action system must satisfy the following properties in order to be well-defined:

(i) For internal actions τ, it must hold that $arity(\tau) = 0$, i.e. internal actions must not have parameters. The same restriction is also placed on STSs in [14].

(ii) The transition relation $\rightarrow$ must contain exactly one definition for each non-internal action, i.e. $\forall \lambda \in \Lambda\colon\ |\{(\lambda, \varphi, \rho) | (\lambda, \varphi, \rho) \in \rightarrow\}| = 1$ must hold.

Although we require that action systems must not contain duplicate observable actions, we allow non-determinism to be expressed through internal actions.

Definition 2.2 [Interpretation of action systems as IOLTSs] Let $\mathcal{AS}$ be an action system given by $\mathcal{AS} = \langle \mathcal{V}, \mathcal{I}, \Lambda_I, \Lambda_U, \iota, \rightarrow \rangle$. Its interpretation $[\![\mathcal{AS}]\!]$ as IOLTS is defined as $[\![\mathcal{AS}]\!] = \langle Q, q_{init}, \Sigma_I, \Sigma_U, \rightarrow_{LTS} \rangle$, where

- $q_{init} = eval \circ \iota$ is the initial state and $Q = \mathfrak{U}^{\mathcal{V}}$ is the set of all states.
- $\Sigma_I = \bigcup_{\lambda \in \Lambda_I} (\{\lambda\} \times \mathfrak{U}^{arity(\lambda)})$ is the set of inputs , $\Sigma_U = \bigcup_{\lambda \in \Lambda_U} (\{\lambda\} \times \mathfrak{U}^{arity(\lambda)})$ is the set of outputs and $\Sigma_\tau = \Sigma_I \cup \Sigma_U \cup \{\tau\}$ is the set of all actions.
- $\rightarrow_{LTS} \subseteq \mathfrak{U}^{\mathcal{V}} \times \Sigma_\tau \times \mathfrak{U}^{\mathcal{V}}$ is defined by the rule:

$$\frac{(\lambda, \varphi, \rho) \in \rightarrow \qquad \varsigma \in \mathfrak{U}^{para(\lambda)} \qquad \vartheta \cup \varsigma \models \varphi \qquad \vartheta' = (\vartheta \cup \varsigma)_{eval} \circ \rho}{(\vartheta, (\lambda, \varsigma(para(\lambda))), \vartheta') \in \rightarrow_{LTS}}$$

However, this definition will not be used in the following, because it would not be possible to utilise the symbolic structure defined by action systems, but it should illustrate action system semantics by means of a well-known formalism. Nevertheless, the fact that it is possible to define semantics for action systems based on Labelled Transition Systems (LTSs) can be used for the development of model-based testing tools, or more specifically for **ioco** checking tools [2,18].

2.3 Symbolic Execution

Differently to [14], symbolic execution trees shall be used to give a symbolic execution semantics for action systems rather than symbolic traces. Nevertheless, some concepts like indexed parameter variables and symbolic states from [14] will

[2] Note that $\mathcal{I}$ is used rather than $\mathcal{P}$ to avoid confusion with power sets and because parameter variables correspond to interaction variables of STSs.

be adapted and used in a similar style. The general idea is to execute actions with symbolic values as parameters and to include the guards of actions executed on a trace in the path condition. Hence, the path condition is a boolean expression over the initial state and symbolic parameters and the symbolic state vector is a mapping from state variables to terms over the initial state and symbolic parameters. Since one action may occur multiple times on a trace, indexed sets of mutually disjoint parameter variables are used in order to distinguish action parameters arising from multiple executions of a single action. The sets $\mathcal{I}_1, \mathcal{I}_2, \ldots$ are used to denote those parameter variables, $\widehat{\mathcal{I}}$ is defined as $\bigcup_j \mathcal{I}_j$. Furthermore, a bijective variable renaming $r_n \in \mathcal{I}_n^{\mathcal{I}}$ is assumed to exist. In the following, it will be assumed that an action system $\mathcal{AS}$ is given, with $\mathcal{AS} = \langle \mathcal{V}, \mathcal{I}, \Lambda_I, \Lambda_U, \iota, \rightarrow \rangle$.

Symbolic States

First, the concept of symbolic states shall be introduced. Like the execution state in the symbolic execution of programs [11], a symbolic state consists of a path condition and a state vector, in which the value of each state variable is described by a term containing symbolic values.

Definition 2.3 [Symbolic States] A symbolic state is a tuple $(\varphi, \rho) \in \mathfrak{F}(\widehat{\mathcal{I}}) \times \mathfrak{T}(\widehat{\mathcal{I}})^{\mathcal{V}}$. The tuple element φ will be referred to as path condition and ρ will be referred to as symbolic state vector. An indexed symbolic state is a tuple $(\varphi, \rho, i) \in \mathfrak{F}(\widehat{\mathcal{I}}) \times \mathfrak{T}(\widehat{\mathcal{I}})^{\mathcal{V}} \times \mathbb{N}_0$, also written as $(\varphi, \rho)_i$ in which all indexed parameter variables occurring in φ and ρ have an index lower than or equal to i.

A symbolic state corresponds to a set of concrete states called interpretations:

Definition 2.4 [Interpretation of Symbolic States] Let $\eta = (\varphi, \rho)$ be a symbolic state. Its interpretation with respect to $\upsilon \in \mathfrak{U}^{\widehat{\mathcal{I}}}$ is defined as $[\![\eta]\!]_\upsilon = \{\upsilon_{eval} \circ \rho | \upsilon \models \varphi\}$. The set of all interpretations $[\![\eta]\!]$ is given by $[\![\eta]\!] = \bigcup_{\upsilon'} [\![\eta]\!]_{\upsilon'}$.

Gaston et al. gave a definition of state inclusion [16], which will be adapted to define equivalence between two symbolic states. Their definition is based on all possible interpretation of symbolic indexed states, but using a slightly different notion of interpretations. Nevertheless, the definition of symbolic state equivalence shall be based on the set of all possible interpretations as well. Hence, two symbolic states $\eta = (\varphi, \rho)$ and $\eta' = (\varphi', \rho')$ are defined to be equivalent, denoted by $\eta \equiv \eta'$, if $[\![\eta]\!] = [\![\eta']\!]$, i.e. they correspond to identical sets of concrete states. A symbolic definition of equivalence is given below.

Definition 2.5 [Equivalence of Symbolic States] Let $\eta = (\varphi, \rho)$ and $\eta' = (\varphi', \rho')$ be two symbolic states. $\eta \equiv \eta'$ iff:

- if for all $\zeta \in \mathfrak{U}^{\mathcal{V}}$ and a $\upsilon \in \mathfrak{U}^{\widehat{\mathcal{I}}}$ such that $\zeta \cup \upsilon \models ((\bigwedge_{x \in \mathcal{V}} x = \rho(x)) \wedge \varphi)$ there exists a $\upsilon' \in \mathfrak{U}^{\widehat{\mathcal{I}}}$ such that $\zeta \cup \upsilon' \models ((\bigwedge_{x \in \mathcal{V}} x = \rho'(x)) \wedge \varphi')$
- and if for all $\zeta' \in \mathfrak{U}^{\mathcal{V}}$ and a $\upsilon'' \in \mathfrak{U}^{\widehat{\mathcal{I}}}$ such that $\zeta' \cup \upsilon'' \models ((\bigwedge_{(x \in \mathcal{V}} x = \rho'(x)) \wedge \varphi')$ there exists a $\upsilon''' \in \mathfrak{U}^{\widehat{\mathcal{I}}}$ such that $\zeta' \cup \upsilon''' \models ((\bigwedge_{x \in \mathcal{V}} x = \rho(x)) \wedge \varphi)$

So in order to determine if two symbolic states η and η' are equivalent, it is necessary to check if both $[\![\eta]\!] \subseteq [\![\eta']\!]$ and $[\![\eta']\!] \subseteq [\![\eta]\!]$ hold.

Quiescence Condition

Since the absence of observations is relevant for the conformance check, a condition for observing quiescence needs to be defined. A state q of an LTS is said to be quiescent if it is not possible to execute an output or an internal action in q [21]. Hence, the following definition for the quiescence condition denoted by Δ, with $\Delta \in \mathfrak{F}(\mathcal{V})$ can be given as in [14]:

$$\Delta \;=\; \bigwedge\{\neg\overline{\exists}_{para(\lambda)}\psi \,|\, \exists\rho : (\lambda,\psi,\rho) \in\rightarrow \text{ with } \lambda \in \Lambda_U \cup \{\tau\}\}$$

The observation of quiescence will be treated like an ordinary action without parameters and denoted by the label δ, i.e. (δ, Δ, id) will be used as quiescence transition. Since the absence of observations does not update the state, the identity function is used as update mapping.

Symbolic Execution Tree

The concept of symbolic execution trees of action systems is inspired by execution trees created for programs [11]. A symbolic execution tree shall, starting from an initial state, encode the effects of symbolically executing arbitrary actions. We chose a tree-based rather than a trace-based description of the symbolic execution of action systems to highlight the effects of actions on symbolic states.

Definition 2.6 [Symbolic Execution Tree of an Action System] Let $Q \subseteq \mathfrak{F}(\widehat{\mathcal{I}}) \times \mathfrak{T}(\widehat{\mathcal{I}})^{\mathcal{V}} \times \mathbb{N}_0$ be a set of indexed symbolic states and let $T \subseteq Q \times (\Lambda_\tau \cup \{\delta\}) \times Q$ be the set of edges of the symbolic execution tree. For $((\varphi,\rho,i),\lambda,(\varphi',\rho',j)) \in T$, the abbreviation $(\varphi,\rho)_i \xrightarrow{\lambda}_T (\varphi',\rho')_j$ will be used. The sets Q and T are defined to be the smallest sets satisfying the following rules:

Initial state:

$$\frac{}{(\top,\iota)_0 \in Q}$$

Execution of actions and quiescence observation:

$$\frac{\begin{array}{ccc}(\varphi,\rho)_n \in Q & (\lambda,\psi,\pi) \;\in \rightarrow \cup \{(\delta,\Delta,id)\} & \lambda \neq \tau \\ \varphi' = \varphi \wedge (\psi[r_{n+1}])[\rho] & \rho' = ([\rho] \circ ([r_{n+1}] \circ \pi))_{\mathcal{V}} & \exists\varsigma \in \mathfrak{U}^{\widehat{\mathcal{I}}} : \varsigma \models \varphi'\end{array}}{(\varphi',\rho')_{n+1} \in Q \qquad (\varphi,\rho)_n \xrightarrow{\lambda}_T (\varphi',\rho')_{n+1}}$$

Execution of internal actions:

$$\frac{\begin{array}{c}(\varphi,\rho)_n \in Q \\ (\tau,\psi,\pi) \;\in\rightarrow \qquad \varphi' = \varphi \wedge \psi[\rho] \qquad \rho' = [\rho] \circ \pi \qquad \exists\varsigma \in \mathfrak{U}^{\widehat{\mathcal{I}}} : \varsigma \models \varphi'\end{array}}{(\varphi',\rho')_n \in Q \qquad (\varphi,\rho)_n \xrightarrow{\tau}_T (\varphi',\rho')_n}$$

Note that the indexes of states directly correspond to the execution depth, at which they have been detected. By convention, we consider states reached by executing internal actions to be at the same depth label as the pre state, so we do not increase the index value for such states. This is allowed by our definition of indexed symbolic states, as internal actions do not have parameters and thereby do not introduce new indexed parameter variables. Although this is also true for the quiescence observation, we increase the symbolic state index as it is observable.

3 Conformance Check

In this section, we will present our approach to conformance checking and test case generation. While the condition for non-conformance is derived from the definition of **sioco** in [14], the implementation for detecting non-conformance follows a similar approach as in [22], which defines a product graph for checking **ioco**. They explore the defined product graph "on the fly" and return a diagnostic sequence of actions leading to a state, where non-conformance may be observed, if the checked IOLTSs are not **ioco**-conform. Analogously, we propose to check for non-conformance by implicitly exploring a symbolic product graph. Beside transitions to system states, the symbolic product graph shall only contain transitions to *fail*-states. As in [22], *fail*-states shall denote that non-conforming behaviour may be observed in the pre state. Each of the *fail*-states consists of the *fail*-label and the condition, which must be satisfied in order to witness non-conformance. Furthermore, the symbolic product graph makes some steps involved in the implementation of the **sioco** check explicit. These include the exploration of the product graph only up to a given depth, the explicit calculation of τ-closures and the execution of actions. Therefore, prior to the definition of the graph some auxiliary functions will be defined.

3.1 Auxiliary Definitions

First, a symbolic τ-closure function shall be defined. It works by calculating a set of symbolic states reachable by executing internal actions.

Definition 3.1 [τ-closure] Let $\mathcal{AS} = \langle \mathcal{V}, \mathcal{I}, \Lambda_I, \Lambda_U, \iota, \rightarrow \rangle$ be an action system, $\kappa \in \mathcal{P}(\mathfrak{F}(\widehat{\mathcal{I}}) \times \mathfrak{T}(\widehat{\mathcal{I}})^{\mathcal{V}})$ be a set of symbolic states. The τ-closure $\tau_{cl}(\kappa)$ of κ is defined by:

$$\begin{aligned} \tau_{cl}\colon & \mathcal{P}(\mathfrak{F}(\widehat{\mathcal{I}}) \times \mathfrak{T}(\widehat{\mathcal{I}})^{\mathcal{V}}) \rightarrow \mathcal{P}(\mathfrak{F}(\widehat{\mathcal{I}}) \times \mathfrak{T}(\widehat{\mathcal{I}})^{\mathcal{V}}) \\ \tau_{cl}(S) &= S \cup \tau_{cl}(\{s | s \in \tau_{next}(S) \wedge \neg\exists s' \in S : s \equiv s'\}) \\ \text{where } \tau_{next}\colon & \mathcal{P}(\mathfrak{F}(\widehat{\mathcal{I}}) \times \mathfrak{T}(\widehat{\mathcal{I}})^{\mathcal{V}}) \rightarrow \mathcal{P}(\mathfrak{F}(\widehat{\mathcal{I}}) \times \mathfrak{T}(\widehat{\mathcal{I}})^{\mathcal{V}}) \\ \tau_{next}(S) &= \bigcup_{(\varphi,\rho)\in S} \left\{ (\varphi \wedge \psi[\rho], [\rho] \circ \pi) \middle| (\tau, \psi, \pi) \in \rightarrow, \exists \varsigma \in \mathfrak{U}^{\widehat{\mathcal{I}}} : \varsigma \models \varphi \wedge \psi[\rho] \right\} \end{aligned}$$

Using this definition, a state will not be explored further if an equivalent state is already in closure. It should be noted that the τ-closure function may be applied on indexed sets of symbolic states in the same way as on non-indexed sets of symbolic states. Since the execution of internal actions does not introduce new parameter variables, the index of a state reached by executing an internal can be chosen to be equal to the index of the pre state.

Since we restrict data types to be finite, it is guaranteed that there is only a finite number of symbolic state equivalence classes, where equivalence is defined as in Definition 2.5. It follows that the τ-closure algorithm terminates after at most n steps, where n is the number of symbolic state equivalence classes.

The product graph will contain product states which are pairs of sets of symbolic states. As for the symbolic states, interpretations of product states as well as an equivalence condition will be given, which will be used for optimisations.

Definition 3.2 [Product States] A product state is a pair of two sets of symbolic states with disjoint sets of state variables, i.e. it is an element of the set $\mathcal{P}(\mathfrak{F}(\widehat{\mathcal{I}}) \times \mathfrak{T}(\widehat{\mathcal{I}})^{\mathcal{V}_P}) \times \mathcal{P}(\mathfrak{F}(\widehat{\mathcal{I}}) \times \mathfrak{T}(\widehat{\mathcal{I}})^{\mathcal{V}_S})$, where $\mathcal{V}_P \cap \mathcal{V}_S = \emptyset$. An indexed product state (κ_i, μ_i) is a pair of sets of indexed symbolic states all sharing some common index i.

Since a product state is intended to be a pair consisting of a set of implementation states and a set of specification states, the definition requires that the state variables of the two sets must be disjoint. In the context of the conformance check, the convention will be used that the left pair element is a set of states of the implementation P, while the right pair element is a set of states of the specification S. For a product state (κ, μ), the path condition is given by $pc((\kappa, \mu)) = \left(\bigvee_{(\gamma_P, \pi_P) \in \kappa} \gamma_P\right) \wedge \left(\bigvee_{(\gamma_S, \pi_S) \in \mu} \gamma_S\right)$. The definition of product state interpretations shall now be given as follows:

Definition 3.3 [Interpretations of Product States] Let (κ, μ) be a product state, its interpretation with respect to υ is defined as $[\![(\kappa, \mu)]\!]_\upsilon = \bigcup_{(\varphi, \rho) \in \kappa} \{\upsilon_{eval} \circ \rho | \upsilon \models \varphi\} \times \bigcup_{(\psi, \pi) \in \mu} \{\upsilon_{eval} \circ \pi | \upsilon \models \psi\}$. All interpretations are given by $[\![(\kappa, \mu)]\!] = \bigcup_\upsilon [\![(\kappa, \mu)]\!]_\upsilon$.

Given two product states κ and κ', the equivalence of κ and κ' will be denoted as $\kappa \equiv_{prod} \kappa'$. Similar to symbolic states, product states are considered to be equivalent, if they correspond to the same sets of concrete states, thus $\kappa \equiv_{prod} \kappa'$ if $[\![\kappa]\!] = [\![\kappa']\!]$. A symbolic condition can be derived analogously.

Execution of Actions

Before a definition of the product graph can be given, two further auxiliary functions need to be defined, which describe how indexed sets of symbolic states are changed through the execution of visible actions. The first function *exec* concerns the actual execution of an action, while the second $exec_{neg}$ concerns the "negated" execution, which is actually used to ignore inputs for performing an *angelic completion* of the implementation, as described for LTSs in [21].

Since the angelic completion adds self-loops for undefined inputs to states of LTSs, $exec_{neg}$ must not perform a state update. As implementations in the context of **sioco** are considered to be weakly input-enabled, $exec_{neg}$ should take into account, that it is not necessary to add self-loops for inputs i if a state may be reached by executing internal actions, in which i is enabled. For this reason, the function expects the disjunction over the guards of all internal actions as third parameter.

$$
\begin{aligned}
&exec : \mathcal{P}(\mathfrak{F}(\widehat{\mathcal{I}}) \times \mathfrak{T}(\widehat{\mathcal{I}})^{\mathcal{V}} \times \mathbb{N}_0) \times \mathfrak{F}(Var) \times \mathfrak{T}(Var)^{\mathcal{V}} \to \mathcal{P}(\mathfrak{F}(\widehat{\mathcal{I}}) \times \mathfrak{T}(\widehat{\mathcal{I}})^{\mathcal{V}} \times \mathbb{N}_0) \\
&exec(\kappa, \psi, \pi) = \{(\varphi \wedge (\psi[r_{i+1}])[\rho], ([\rho] \circ ([r_{i+1}] \circ \pi))_{\mathcal{V}})_{i+1} | \\
&\qquad\qquad (\varphi, \rho)_i \in \kappa \wedge \exists \varsigma \in \mathfrak{U}^{\widehat{\mathcal{I}}} : \varsigma \models \varphi \wedge (\psi[r_{i+1}])[\rho]\} \\
&exec_{neg} : \mathcal{P}(\mathfrak{F}(\widehat{\mathcal{I}}) \times \mathfrak{T}(\widehat{\mathcal{I}})^{\mathcal{V}} \times \mathbb{N}_0) \times \mathfrak{F}(Var) \times \mathfrak{F}(Var) \to \mathcal{P}(\mathfrak{F}(\widehat{\mathcal{I}}) \times \mathfrak{T}(\widehat{\mathcal{I}})^{\mathcal{V}} \times \mathbb{N}_0) \\
&exec_{neg}(\kappa, \psi, \zeta) = \{(\varphi \wedge \neg(\psi[r_{i+1}])[\rho] \wedge \neg\zeta[\rho], \rho)_{i+1} | (\varphi, \rho)_i \in \kappa \wedge \\
&\qquad\qquad \exists \varsigma \in \mathfrak{U}^{\widehat{\mathcal{I}}} : \varsigma \models \varphi \wedge \neg(\psi[r_{i+1}])[\rho] \wedge \neg\zeta[\rho]\}
\end{aligned}
$$

3.2 Product Graph

The product graph for two action systems $\mathcal{AS}_P$ and $\mathcal{AS}_S$ shall now be defined.

Definition 3.4 [Product graph] Let $\mathcal{AS}_S = \langle \mathcal{V}_S, \mathcal{I}, \Lambda_I, \Lambda_U, \iota_S, \rightarrow_S \rangle$ be an action system representing a specification, let $\mathcal{AS}_P = \langle \mathcal{V}_P, \mathcal{I}, \Lambda_I, \Lambda_U, \iota_P, \rightarrow_P \rangle$ be an action system representing an implementation and let $d \in \mathbb{N}_0$ be the maximum exploration depth. The deterministic symbolic synchronous product graph $\mathcal{AS}_P \times_{sioco_{det}} \mathcal{AS}_S(d)$ bounded by d is a tuple $SP = \langle Q, q_{init}, \rightarrow_{SP}, \Lambda_I, \Lambda_U \rangle$ where $Q \subseteq \mathcal{P}(\mathfrak{F}(\widehat{\mathcal{I}}) \times \mathfrak{T}(\widehat{\mathcal{I}})^{\mathcal{V}_P} \times \mathbb{N}_0) \times \mathcal{P}(\mathfrak{F}(\widehat{\mathcal{I}}) \times \mathfrak{T}(\widehat{\mathcal{I}})^{\mathcal{V}_S} \times \mathbb{N}_0)$, $\Lambda_\delta = \Lambda_I \cup \Lambda_U \cup \{\delta\}$, $\rightarrow_{SP} \subseteq Q \times \Lambda_\delta \times (Q \cup (\{fail\} \times \mathfrak{F}(\widehat{\mathcal{I}} \cup \mathcal{I})))$ and $q_{init} = (\tau_{cl}(\{(\top, \iota_P)_0\}), \tau_{cl}(\{(\top, \iota_S)_0\}))$. For $(q, \lambda, q') \in \rightarrow_{SP}$, the abbreviation $q \xrightarrow{\lambda}_{SP} q'$ will be used. The transition relation $\rightarrow_{SP}$ and the set Q are defined as the smallest sets, satisfying the following rules:

Initial state:

$$\frac{}{q_{init} \in Q}$$

Observations:

$$\frac{\begin{array}{c}(\kappa_i, \mu_i) \in Q \qquad i < d \qquad \lambda \in \Lambda_U \qquad (\lambda, \varphi_P, \rho_P) \in \rightarrow_P \cup \{(\delta, \Delta_P, id)\} \\ (\lambda, \varphi_S, \rho_S) \in \rightarrow_S \cup \{(\delta, \Delta_S, id)\} \qquad \kappa_{i+1} = \tau_{cl}(exec(\kappa_i, \varphi_P, \rho_P)) \\ \mu_{i+1} = \tau_{cl}(exec(\mu_i, \varphi_S, \rho_S)) \qquad \exists \varsigma \in \mathfrak{U}^{\widehat{\mathcal{I}}} : \varsigma \models pc((\kappa_{i+1}, \mu_{i+1}))\end{array}}{(\kappa_{i+1}, \mu_{i+1}) \in Q \qquad (\kappa_i, \mu_i) \xrightarrow{\lambda}_{SP} (\kappa_{i+1}, \mu_{i+1})}$$

Inputs:

$$\frac{\begin{array}{c}(\kappa_i, \mu_i) \in Q \qquad i < d \qquad \lambda \in \Lambda_I \qquad (\lambda, \varphi_P, \rho_P) \in \rightarrow_P \qquad (\lambda, \varphi_S, \rho_S) \in \rightarrow_S \\ \kappa_{i+1} = \tau_{cl}(exec(\kappa_i, \varphi_P, \rho_P) \cup exec_{neg}(\kappa_i, \varphi_P, \zeta)) \qquad \zeta = \bigvee_{(\tau, \gamma, \pi) \in \rightarrow_P} \gamma \\ \mu_{i+1} = \tau_{cl}(exec(\mu_i, \varphi_S, \rho_S)) \qquad \exists \varsigma \in \mathfrak{U}^{\widehat{\mathcal{I}}} : \varsigma \models pc((\kappa_{i+1}, \mu_{i+1}))\end{array}}{(\kappa_{i+1}, \mu_{i+1}) \in Q \qquad (\kappa_i, \mu_i) \xrightarrow{\lambda}_{SP} (\kappa_{i+1}, \mu_{i+1})}$$

Detection of non-conformance:

$$\frac{\begin{array}{c}(\kappa_i, \mu_i) \in Q \qquad i \leq d \qquad \lambda \in \Lambda_U \cup \{\delta\} \qquad (\lambda, \varphi_P, \rho_P) \in \rightarrow_P \cup \{(\delta, \Delta_P, id)\} \\ (\lambda, \varphi_S, \rho_S) \in \rightarrow_S \cup \{(\delta, \Delta_S, id)\} \qquad (\chi, \eta)_i \in \mu_i \qquad \exists \varsigma \in \mathfrak{U}^{\widehat{\mathcal{I}} \cup \mathcal{I}} : \varsigma \models \xi\end{array}}{(\kappa_i, \mu_i) \xrightarrow{\lambda} (fail, \xi)}$$

$$\text{where } \xi = \left(\bigvee_{(\gamma_P, \pi_P) \in \kappa_i} \gamma_P \wedge \varphi_P[\pi_P]\right) \wedge \chi \wedge \neg \left(\bigvee_{(\gamma_S, \pi_S) \in \mu_i} \gamma_S \wedge \varphi_S[\pi_S]\right)$$

The implementation $\mathcal{AS}_P$ is not **sioco**-conform to $\mathcal{AS}_S$ if there exists a path from q_{init} to a *fail*-state. Since **sioco** coincides with **ioco** [14] and our condition for non-conformance is based on the original definition of **sioco**, the product graph contains a *fail*-state iff there exists a suspension trace of $[\![\mathcal{AS}_S]\!]$ of length $\leq d$, which shows that $[\![\mathcal{AS}_P]\!]$ ~~**ioco**~~ $[\![\mathcal{AS}_S]\!]$.

There are a few things to note about the product graph. The condition, which must be satisfied in order to be able to observe quiescence is denoted as Δ_P or Δ_S rather than Δ, because the condition depends on whether the implementation or the specification is observed. Hence, Δ_P denotes that the condition is formed based on the transitions in $\rightarrow_P$, while Δ_S is derived using $\rightarrow_S$.

As noted before, action systems may behave non-deterministically. The **sioco** conformance relation, takes non-determinism into account as well. Therefore, the

conformance check is able to handle action systems containing internal actions and does not produce spurious counterexamples, i.e. it does not identify action systems to be non-conforming, which are actually conforming.

Since we intend to use mutated specification models as implementations, we can not guarantee input-enabledness of implementations, which is a precondition for **sioco**-conformance check. Hence, differently from the definition of **sioco** [14], implementations can not be considered to be weakly input-enabled, an angelic completion is rather performed to make implementations input-enabled.

Furthermore, the product graph allows actions to be executed only if the path condition of the target product state, which contains both implementation and specification states, is satisfiable, while **ioco** is defined for suspension traces of the specification. This restriction is used, as the non-conformance condition given below would not be satisfiable anyway for product states with unsatisfiable path conditions. The non-conformance condition corresponds to the negation of the condition for **sioco**-conformance given in [14].

Definition 3.5 [Non-conformance Condition for Product States] Let $\mathcal{AS}_S$, $\mathcal{AS}_P$ and $SP = \langle Q_{SP}, q_{init}, \rightarrow_{SP}, \Lambda_I, \Lambda_U \rangle$ be defined as in Definition 3.4, The non-conformance condition for a product state $(\kappa, \mu) \in Q_{SP}$ is given by:

$\mathcal{AS}_P \ \mathbf{\not{sioco}}\ \mathcal{AS}_S$ if:

$$\exists \lambda \in \Lambda_U \cup \{\delta\} : \overline{\exists}_{\widehat{\mathcal{I}} \cup \mathcal{I}} \left(\bigvee_{(\gamma_P, \pi_P) \in \kappa} \gamma_P \wedge \varphi_P[\pi_P] \right) \wedge \chi \wedge \neg \left(\bigvee_{(\gamma_S, \pi_S) \in \mu} \gamma_S \wedge \varphi_S[\pi_S] \right)$$

where $\exists \pi : (\lambda, \varphi_P, \pi) \in \rightarrow_P \cup \{(\delta, \Delta_P, id)\}$ and $\exists \pi : (\lambda, \varphi_S, \pi) \in \rightarrow_S \cup \{(\delta, \Delta_S, id)\}$ and $\exists \rho : (\chi, \rho) \in \mu$

Test Case Generation

The implementation of the test case generator performs a depth-first search for unsafe states in the product graph. Following the definition given in [5], unsafe states are states in which non-conforming behaviour may be observed. In our case, these are product states which fulfil the condition given in Definition 3.5. If such an unsafe state is found, a symbolic test case is returned, which is a pair formed of:

- a sequence of actions leading to this state
- and the non-conformance condition for this state. However, the condition associated with a test case does not contain an existential quantification over outputs, but is fixed for some observation, for which it is satisfiable.

During test case execution we use both the sequence of actions and the non-conformance condition to direct the execution to a potentially unsafe concrete state. In case of non-determinism, the sequential test case is rather a test purpose than a test case [4]. It either has to be augmented with inconclusive verdicts as in [6] or extended to a branching adaptive test case as in [2,3].

4 Optimisations

We developed several optimisations to decrease the run-time of the search in the product graph, which will be discussed in the following. Some of them specifically target model-based mutation using first-order mutants. As such, they make use of

an efficient syntactical mutation analysis inspired by [5], which detects the location of mutations. We distinguish mutations of the init block, the state update of an action and the guard of an action. In general, an algorithm applying the given optimisations will not detect fewer conformance violations. One optimisation, however, breaks this rule in order to generate a more comprehensive test suite.

4.1 Precomputation on Specification

Since during test case generation, a variety of mutants is checked against one specification, it is advantageous to precompute the action traces executable by the specification. Hence, the symbolic execution tree is created up to the maximum search depth d. This step itself can be optimised using Definition 2.5 and the observation that a symbolic state does not need to be explored if an equivalent state has already been explored, because the same set of action will be enabled for the same set of parameters in both states. Hence, the exploration is stopped if some state is reached, for which there exists an equivalent and already explored state. This introduces loops, such that actually a directed symbolic execution graph is created. By applying this approach, the symbolic execution graph is guaranteed to be of finite size as all available data types are finite, which ensures that the number of symbolic state equivalence classes is finite.

Based on the symbolic execution tree, further data structures can be created:

- The set of all equivalence classes of symbolic states in the symbolic execution tree defined by the equivalence relation $\equiv$, where the symbolic state explored first is chosen as canonical representative of the corresponding equivalence class
- A table which lists sets of enabled actions reachable within a given number of steps from canonical representatives of symbolic state equivalence classes

To be able to use these data structures, every symbolic state of the specification is associated with a symbolic graph node.

4.2 Early Stopping of Product Graph Exploration

The exploration of the product graph may be stopped before reaching the maximum depth as well, which can be done by utilising equivalence of product states. Intuitively, the post states of equivalent product states will be equivalent as well, thus if non-conformance is not detected following the exploration of a product state q, it will not be detected by exploring some product state equivalent to q. Hence, we can define two simplifications: (1) if a state q equivalent to the current product state has already been explored, it is not necessary to perform a non-conformance check; (2) if q was explored at lower or equal depth, the search can be stopped.

Checking of product state equivalence is however computationally intensive, so an approximation of the equivalence check was implemented. Given two products states (κ, μ) and (κ', μ'), in many cases where $(\kappa, \mu) \equiv_{prod} (\kappa', \mu')$, the symbolic states in μ will be equivalent to states in μ'. Since μ and μ' are states of the specification, it is possible to efficiently check if for all states in μ there exist corresponding equivalent states in μ' by utilising the precomputed symbolic state equivalence classes. Consequently $(\kappa, \mu) \equiv_{prod} (\kappa', \mu')$ is only checked, if this condition holds. Otherwise the product states are considered to be inequivalent.

4.3 Restriction of Angelic Completion

The angelic completion may lead to cases of non-conformance, which do not involve mutated actions, i.e. the angelic completion of a specification may be non-conforming to the specification itself. This may lead to the generation of a large number of equivalent test cases, if it is possible to reach a conformance violation caused by angelic completion by a low number of steps and thereby without executing mutated actions. Therefore, the angelic completion is not performed until the mutated action is executed. Another reason for this decision is that angelic completion may be seen as an additional mutation and we focus on first-order mutants.

4.4 Exploiting Syntactical Mutation Analysis

As a result of the restriction of angelic completion, the states of the implementation and the specification are identical until the mutated action is executed. Building upon this observation, further optimisations can be implemented:

- As long as the mutated action does not need to be executed, the symbolic execution graph can be used for the mutant as well. If the mutation affects the guard of an internal or an output action, the observation of quiescence must be performed explicitly for the mutant.
- Product states reached without executing the mutation are of the form (κ, κ), which allows further simplification of the product state equivalence check.
- As long as the mutated action has not been executed, at most two non-conformance checks need to be performed at each step. The actual number depends on whether the mutation affects the guard of an internal or output action. If it is not possible to reach the mutation along the search path, the search may be stopped before hitting the maximum search depth.

4.5 Checking if Input Guard Weakened

The last optimisation is only applicable for mutants generated from specifications without internal actions by mutating the guard of an input action. Nevertheless, since it detects equivalent mutants using only one check, it can significantly increase the performance of the conformance check. This optimisation is based on the fact that **ioco** allows implementation freedom for non-specified inputs [20]. The key insight to this optimisation is that a mutant conforms to the specification, if the mutation affects only the guard of an input action and the mutant accepts all specified inputs. Consequently it is possible to infer that a given mutant is conforming, by positively checking that the mutation weakens the guard of an input action.

5 Case Study

We implemented the presented approach to test case generation using Microsoft's Satisfiability Modulo Theories (SMT)-solver Z3 (v4.3.2.)[12] and first experiments have been carried out. In this section, we compare the run-time of the **sioco**-based implementation with the run-time of an **ioco**-based implementation used in [18]. The latter uses IOLTS-semantics of action systems and works by explicitly enumerating all possible actions in every visited state. All experiments were performed on

	conformance-check		test case generation	
	sioco	**ioco**	**sioco**	**ioco**
mean	0.44	34.97	~ 0	0.5
median	0.03	3.44	~ 0	0.48
max	31.57	2.96 h	0.02	1.52
min	~ 0	0.03	~ 0	~ 0

Table 1
The execution times for the two most computationally intensive steps performed during test cases generation, for both the sioco-based and the ioco-based implementation. All values are given in seconds, unless otherwise noted.

the same PC equipped with 8 GB RAM and an Intel i7 quad-core processor running at 3.4 GHz. In both experiments however only one core was used. The run-time measurements for the **ioco**-based implementation have also been used in [18].

The model used for the comparison specifies the behaviour of a device measuring particle counts in exhaust gas and contains 69 input and 20 output actions. It was derived from the model used in [6], which was defined using a different action system language. Since this language allows for nested guarded commands in actions, the original model contains a lower number of actions.

The results are listed in Table 1. It should be noted that the **sioco**-based implementation needs approximately 209 seconds for precomputation, while the **ioco**-based approach does not perform any precomputation. However, this additional effort pays off, because the difference in overall execution time is about 416 minutes, with the symbolic implementation being faster, although it processes about two and a half times as many mutants. Despite the fact that both implementations use different input languages, the results are comparable as the same system is modelled, similar mutation operators are used and the mean and median exploration depth necessary to detect non-conformance are approximately the same for both approaches. Moreover, the relative amount of conforming (equivalent) mutants is about 22 per cent for both approaches. As there is no direct correspondence between mutants and because of the different nature of both approaches, the structure of the mutants leading to the maximum run-times for the conformance check is not similar. In fact, the mutant leading to a run-time of 2.96 hours for the **ioco**-based implementation would be detected to be conforming using only one check by the symbolic approach, because the mutant weakens the guard of an input action.

6 Conclusion

We presented an approach to use symbolic input output conformance checking for test case generation and gave guidelines on how to implement the conformance check efficiently. The optimisations are targeted towards model-based mutation testing, which creates test cases covering faults corresponding to model mutations. Applying angelic completion to the mutant, we implicitly cover another class of faults: as the angelic completion essentially ignores non-specified inputs, our test case generation strategy covers faults corresponding to ignored inputs as well.

To our knowledge, we have implemented the first fully symbolic **ioco**-conformance checker. However, the **sioco**-conformance relation has already been used as a conformance relation for model-based online testing [13]. In contrast to

their work, we rather focus on test case generation than on execution, which is performed randomly and on-the-fly in [13]. An adapted version of **ioco** is also used in conjunction with symbolic specifications in [16]. Gaston et al. present an algorithm for testing SUTs based on finite behaviours of the specification selected by test purposes. Their notion of test purposes can be related to our test case selection strategy. However, while the test cases generated by our approach are specified through selected finite behaviours as well, they also contain a condition to further restrict the set of allowed action parameters. Nevertheless, test case execution based on action systems is performed similarly, but is not addressed here.

Actually, the technique of using test purposes to select test cases has originally been proposed by Jard and Jéron [17] to derive test cases for ioco-based conformance testing. Mutants can be seen as test purposes, as has been shown by Aichernig and Corrales Delgado [4]. The test case generation strategy can be compared to the strategy applied in [2,3], which is based on **ioco** and also uses action systems. However, the conformance checker used in [2,3] searches for all unsafe states of a given mutant and works concretely, while at the time of writing this paper the symbolic checker stops searching after finding the first unsafe state.

Since we base our work on the symbolic framework developed for IOSTS [14], the product graph and thereby the test case generation process may be adapted to support IOSTSs. First experiments like the aforementioned case study have shown that it is a promising approach. However, further experiments have shown that our approach to tackle the path explosion problem [11] by utilising equivalence of states may lead to poor performance for more complex models. Hence, alternative non-complete search strategies may be better suited if equivalence checks are intractable.

Although we focus on test case generation based on first-order mutants, the conformance check may be used for arbitrary action systems by disabling optimisations, which assume some specific syntactic structure. Hence, it is possible to use symbolic execution to test whether a concrete specification conforms to an abstract specification. Conformance verification by testing has also been targeted in [15], but using a combination of concrete and symbolic execution and using refinement.

In the next step, we plan to carry out further experiments and to implement a translation from the action systems language used in [6] to the language used in this paper in order to alleviate modelling. Furthermore, the effectiveness of test case execution needs to be considered by subsequent work.

In our current applications, we can safely assume a synchronous communication between test driver and SUT. However, this assumption may not hold in general [19]. To circumvent this problem, we could compose our models with action systems modelling message queues prior to the conformance check. Such an approach would be able to faithfully model asynchronous communication performed during testing and would not result in state space explosion, as we check conformance symbolically. Hence, it would be interesting to study the composition of action systems.

6.0.1 Acknowledgments.
Research herein was funded by the Austrian Research Promotion Agency (FFG), project number 845582, Trust via cost function driven model based test case generation for non-functional properties of systems of systems (TRUCONF).

References

[1] Abrial, J.-R., "Modeling in Event-B: System and Software Engineering," Cambridge University Press, New York, NY, USA, 2010, 1st edition.

[2] Aichernig, B. K., H. Brandl, E. Jöbstl and W. Krenn, *Model-based Mutation Testing of Hybrid Systems*, in: F. S. de Boer, M. M. Bonsangue, S. Hallerstede and M. Leuschel, editors, *Formal Methods for Components and Objects - 8th International Symposium, FMCO 2009, Eindhoven, The Netherlands, November 4-6, 2009. Revised Selected Papers*, Lecture Notes in Computer Science **6286** (2010), pp. 228–249.

[3] Aichernig, B. K., H. Brandl, E. Jöbstl, W. Krenn, R. Schlick and S. Tiran, *Killing strategies for model-based mutation testing*, Software Testing, Verification and Reliability (2014), article first published online: 3 FEB 2014.

[4] Aichernig, B. K. and C. C. Delgado, *From faults via test purposes to test cases: on the fault-based testing of concurrent systems*, in: L. Baresi and R. Heckel, editors, *Fundamental Approaches to Software Engineering*, Lecture Notes in Computer Science **3922**, Springer Berlin Heidelberg, 2006 pp. 324–338.

[5] Aichernig, B. K. and E. Jöbstl, *Efficient refinement checking for model-based mutation testing*, in: *QSIC 2012, 12th International Conference on Quality Software, Xi'an, Shaanxi, China, August 27-29*, 2012, pp. 21–30.

[6] Aichernig, B. K., E. Jöbstl and S. Tiran, *Model-based mutation testing via symbolic refinement checking*, Science of Computer Programming **97** (2015), pp. 383–404.

[7] Back, R.-J. and R. Kurki-Suonio, *Decentralization of process nets with centralized control*, Distributed Computing **3** (1989), pp. 73–87.

[8] Back, R. J. R. and F. Kurki-Suonio, *Distributed cooperation with action systems*, ACM Transactions on Programming Languages and Systems (TOPLAS) **10** (1988), pp. 513–554.

[9] Bonsangue, M. M., J. N. Kok and K. Sere, *An approach to object-orientation in action systems.*, in: J. Jeuring, editor, *Mathematics of Program Construction*, Lecture Notes in Computer Science **1422** (1998), pp. 68–95.

[10] Butler, M. J., *Stepwise refinement of communicating systems*, Science of Computer Programming **27** (1996), pp. 139 – 173.

[11] Cristian Cadar, K. S., *Symbolic execution for software testing: Three decades later*, Communications of the Association for Computing Machinery (CACM 2013) **56** (2013), pp. 82–90.

[12] De Moura, L. and N. Bjørner, *Z3: An efficient SMT solver*, in: *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08 (2008), pp. 337–340.

[13] Frantzen, L., M. de las Nieves Huerta, Z. G. Kiss and T. Wallet, *On-the-fly model-based testing of web services with Jambition*, in: *Web Services and Formal Methods, 5th International Workshop, WS-FM 2008, Milan, Italy, September 4-5, 2008, Revised Selected Papers*, 2008, pp. 143–157.

[14] Frantzen, L., J. Tretmans and T. A. C. Willemse, *A symbolic framework for model-based testing*, in: *Proceedings of the First Combined International Conference on Formal Approaches to Software Testing and Runtime Verification*, FATES'06/RV'06 (2006), pp. 40–54.

[15] Gall, P. L., N. Rapin and A. Touil, *Symbolic execution techniques for refinement testing.*, in: Y. Gurevich and B. Meyer, editors, *TAP*, Lecture Notes in Computer Science **4454** (2007), pp. 131–148.

[16] Gaston, C., P. L. Gall, N. Rapin and A. Touil, *Symbolic execution techniques for test purpose definition.*, in: M. U. Uyar, A. Y. Duale and M. A. Fecko, editors, *TestCom*, Lecture Notes in Computer Science **3964** (2006), pp. 1–18.

[17] Jard, C. and T. Jéron, *TGV: theory, principles and algorithms*, International Journal on Software Tools for Technology Transfer **7** (2005), pp. 297–315.

[18] Jöbstl, E., "Model-Based Mutation Testing with Constraint and SMT Solvers," Ph.D. thesis, Graz University of Technology, Institute for Software Technology (2014).

[19] Simao, A. and A. Petrenko, *Generating asynchronous test cases from test purposes*, Information & Software Technology **53** (2011), pp. 1252–1262.

[20] Tretmans, J., *Test generation with inputs, outputs and repetitive quiescence.*, Software - Concepts and Tools **17** (1996), pp. 103–120.

[21] Tretmans, J., *Model based testing with labelled transition systems*, in: *Formal Methods and Testing, An Outcome of the FORTEST Network, Revised Selected Papers*, Lecture Notes in Computer Science **4949** (2008), pp. 1–38.

[22] Weiglhofer, M. and F. Wotawa, *"On the fly" input output conformance verification*, in: *Proceedings of the IASTED International Conference on Software Engineering*, SE '08 (2008), pp. 286–291.