Model-Based Testing and Some Steps towards Test-Based Modelling

Jan Tretmans[1,2,⋆]

[1] Embedded Systems Institute, Eindhoven, The Netherlands
[2] Radboud University, Institute for Computing and Information Sciences, Nijmegen, The Netherlands
jan.tretmans@esi.nl

Abstract. Model-based testing is one of the promising technologies to increase the efficiency and effectiveness of software testing. In model-based testing, a model specifies the required behaviour of a system, and test cases are algorithmically generated from this model. Obtaining a valid model, however, is often difficult if the system is complex, contains legacy or third-party components, or if documentation is incomplete. Test-based modelling, also called automata learning, turns model-based testing around: it aims at automatically generating a model from test observations. This paper first gives an overview of formal, model-based testing in general, and of model-based testing for labelled transition system models in particular. Then the practice of model-based testing, the difficulty of obtaining models, and the role of learning are discussed. It is shown that model-based testing and learning are strongly related, and that learning can be fully expressed in the concepts of model-based testing. In particular, test coverage in model-based testing and precision of learned models turn out to be two sides of the same coin.

Keywords: model-based testing, test-based modelling, automata learning.

1 Introduction

Testing is an experimental way to check whether a (software) system does what it should do. Experiments, i.e., test cases, are applied to check whether the system under test (SUT) behaves as expected and prescribed in its specification and requirements documents. Systematic testing plays an important role in the demand for improved quality of systems and software. Testing, however, is often a manual and laborious process without effective automation, which makes it error-prone, time consuming, and very costly. Estimates are that testing takes 30-50% of the total software development effort. This leads to the quest for more effective and more efficient testing.

⋆ This work has been supported by the EU FP7 under grant number ICT-214755: Quasimodo.

Model-based testing is a promising new technology that can contribute to increasing the efficiency and effectiveness of the testing process. In model-based testing, a model is the starting point for testing. This model expresses precisely and completely what the SUT should do, and should not do, and, consequently, it is a good basis for systematically generating test cases. Model-based testing makes it possible to generate an efficient set of test cases, including test oracles, completely automatically from a model of required SUT behaviour. In this way, model-based testing allows for test automation that goes well beyond the mere automatic execution of manually crafted test scripts, which is the current state of practice. And if the model is valid, i.e., it expresses the system requirements accurately, then all these algorithmically generated tests are provably valid, too.

In Sect. 2, this paper first discusses the ideas, concepts, ingredients, and requirements of model-based testing in general, both informally and in a more rigorous framework. Then Sect. 3 presents a specific theory for model-based testing called the **ioco** approach, where models are expressed as labelled transition systems, and correctness of an SUT with respect to its model is expressed with the **ioco**-implementation relation. This approach provides a well-defined foundation for model-based testing, and it has proved to be a good basis for several practical model-based test generation tools and their application. Sect. 2 and 3 are mainly based on [43], and are intended to give an overview of formal, model-based testing in general, and the **ioco** approach in particular.

Model-based testing starts with a model that is presumed to be correct and valid. Obtaining or constructing a valid model, however, may be difficult if the system is complex, if specifiers and designers do not make models, if the system includes legacy or third-party components, or if documentation is missing or incomplete. The emerging area of *automata learning* or *test-based modelling* aims at generating models automatically from observations made during testing, following a kind of black-box reverse engineering approach [4,37,23,34,6,49,8,36,1,40,39,3]. Sect. 4 starts with discussing some practical model-based testing issues and projects, and how learning plays a role therein although in an informal and ad-hoc way. The second part of Sect. 4 discusses learning in a more systematic way, in particular placing learning in the context of the model-based testing framework of Sect. 2 and the **ioco**-test theory of Sect. 3. Sect. 4 does not intend to give an overview of learning, nor does it present any learning algorithms. It does discuss learning of nondeterministic systems and it considers the consequences of dropping the equivalence requirement between learned model and teaching system, and compares this approach with the currently prevailing Angluin-style of learning [4], such as in the learning tool environment LearnLib [40].

Classification of Model-Based Testing. There are different kinds of model-based testing depending on the kind of models being used, the quality aspects being tested, the level of formality involved, and the degree of accessibility and observability of the system being tested. In this contribution we consider model-based testing as *formal*, *specification-based*, *active*, *black-box*, *functionality testing.*

It is *testing*, because it involves checking some properties of the SUT by systematically performing experiments on the real, executing SUT, as opposed to, e.g., formal verification, where properties are checked on the level of formal descriptions of the system. The kind of properties being checked are concerned with *functionality*, i.e., testing whether the system correctly does what it should do in terms of correct responses to given stimuli, as opposed to, e.g., performance, usability, reliability, or security properties. Such classes of properties are also referred to as quality characteristics. The testing is *active*, in the sense that the tester controls and observes the SUT in an active way by giving stimuli and triggers to the SUT, and observing its responses, as opposed to passive testing, or monitoring.

The basis and starting point for testing is the *specification*, which prescribes what the SUT should, and should not do. The specification is given in the form of some model of behaviour to which the behaviour of the SUT must conform. This model is assumed to be correct and valid: it is not itself the subject of testing or validation. Moreover, the testing is *black-box*. The SUT is seen as a black box without internal detail, which can only be accessed and observed through its external interfaces, as opposed to white-box testing, where the internal structure of the SUT, i.e., the code, is the basis for testing.

Finally, we deal with *formal testing*: the model, or specification, prescribing the desired behaviour is given in some formal language with precisely defined syntax and semantics. But formal testing involves more than just a formal specification. It also involves a formal definition of what a conforming SUT is, a well-defined algorithm for the generation of tests, and a correctness proof that the generated tests are sound and exhaustive, i.e., that they exactly test what they should test.

2 Model-Based Testing

In model-based testing there is a *system under test* (SUT), a *model* that serves as *specification*, and the question whether the behaviour of the SUT *conforms to* the behaviour expressed in the model. To check conformance *test cases* are constructed from the model through *test generation* and *selection*. *Test execution* and *analysis* of test results leads to a *verdict* whether the SUT indeed conforms to the model.

These ingredients of model-based testing are now discussed in general, both informally and in a more rigorous way. The next section will elaborate these for a particular model-based testing approach using labelled transition systems as models and **ioco** as notion of conformance.

System Under Test. The system to be tested is called the *system under test* (SUT). The SUT is a real, physical object, such as a piece of hardware, a computer program with all its libraries running on a particular processor, an embedded system consisting of software embedded in some physical device, or a process control system with sensors and actuators. The SUT is treated as a black

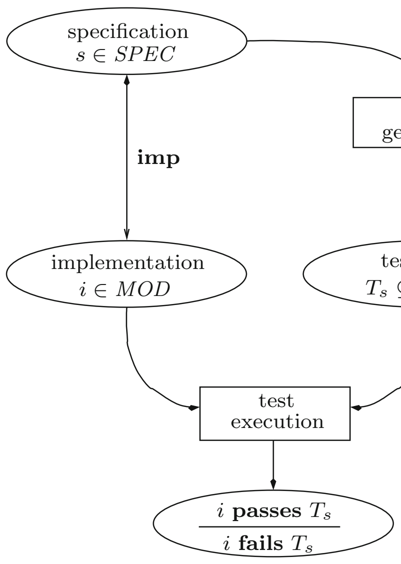


Fig. 1. The process of model-based testing

box exhibiting behaviour. A tester can only control and observe the SUT via its external interfaces, where stimuli and inputs can be provided and responses and outputs are observed. Identifying these test interfaces of the SUT, in terms of, e.g., ports, programming interfaces, message exchanges, or communication lines, is an important first step for (model-based) testing. The occurrence of input and output actions on these interfaces, together with their inter-dependencies and ordering, constitutes the behaviour of the SUT. It is this behaviour of the SUT that will be tested.

Model. The second main ingredient for model-based testing is the specification *model*. The model specifies which behaviours are allowed and which are forbidden. In our formal, model-based testing approach the model is expressed in some formal language, i.e., a language with a formal syntax and semantics. Let this language, i.e., the set of all valid expressions in this language, be denoted by $SPEC$, then a specification model s is an element of this language: $s \in SPEC$.

Conformance. The goal of model-based testing is to check whether the actual behaviour of the SUT conforms to the behaviour expressed in the model. To relate an SUT to a model, the first, static step is to map the real interfaces, inputs, and outputs of the SUT, to their abstract descriptions in the model, e.g., to map the concrete socket connection $\langle 192.168.1.1, 7890\rangle$ to the abstract port p used in the model, and to map the concrete message with bit pattern 01010101 to the abstract message `Init`.

The second, dynamic step of relating a model to an SUT is stating precisely when an SUT correctly implements the behaviour described in a model

$s \in SPEC$. An *implementation relation*, or *conformance relation*, defines the conditions under which the behaviour of an SUT complies with the behaviour prescribed in s. Such a relation is necessary because s in itself does not completely define which SUT behaviours are correct, e.g., whether the SUT *may* or *must* implement all behaviours described in s.

An implementation relation typically answers such questions, but, if we want to define such a formal implementation relation between SUTs and specifications, we encounter a problem. Whereas a specification s is a formal object taken from the formal domain $SPEC$, an SUT is not amenable to formal reasoning. An SUT is not a formal object: it is a real, physical thing, existing in the world of material objects, on which only experiments and tests can be performed.

In order to formally reason about SUTs we do a little trick: we make the assumption that any real SUT can be modelled by some formal object i_{SUT} in a domain of models MOD. The domain MOD is a-priori chosen, may be different from $SPEC$, and is referred to as the universe of implementation models. This assumption is commonly referred to as the *test assumption* or *test hypothesis* [7,20]. Note that the test assumption presupposes a particular domain of models MOD, and that it is only assumed that a valid model i_{SUT} of the SUT exists in this domain, but not that this model i_{SUT} is a-priori known.

Thus, the test assumption allows reasoning about SUTs as if they were formal objects in MOD. This is what we will do from now on, and we call such a formal SUT model an implementation. Consequently, conformance is expressed by a formal relation between implementations and specifications, and that relation is called the *implementation relation* denoted by $\mathbf{imp} \subseteq MOD \times SPEC$. An implementation $i \in MOD$ is said to be correct with respect to $s \in SPEC$ if $i \;\mathbf{imp}\; s$. Implementation relations for labelled transition systems are further discussed in Sect. 3.

Test Cases. A *test case* specifies the experiment that is performed on the SUT. It specifies the inputs, or stimuli, to be supplied to the SUT, the outputs, or responses, expected from the SUT, and the ordering of these inputs and outputs. Formally, we assume a domain of test cases $TEST$ from which test cases are taken.

Test Execution and Analysis. We use the term *test execution* for applying a test case to an SUT, resulting in some observations. To execute a test case, the abstract actions of the test case must invoke the concrete interfaces of the SUT, and the concrete observations made on the SUT must be interpreted in terms of the abstract test case actions. This is called *adaptation*, and the component in the test execution environment taking care of this is usually called the *adapter*. Formally, the process of executing a test case $t \in TEST$ against an SUT is denoted by $\text{EXEC}(t, \text{SUT})$.

During test execution a number of observations is made, e.g., occurring events will be logged, or the response of the implementation to a particular stimulus will be recorded. Let there be a domain of observations OBS, then test execution, due to possible nondeterminism, leads to a set of observations: $\text{EXEC}(t, \text{SUT}) \subseteq OBS$.

Test execution $\text{EXEC}(t, \text{SUT})$ is not a formal concept but corresponds to the physical execution of a test case. This process is lifted to the level of formal models by introducing an observation function $obs : \mathit{TEST} \times \mathit{MOD} \rightarrow \mathcal{P}(\mathit{OBS})$ (where $\mathcal{P}(\mathit{OBS})$ denotes the power set of OBS, i.e., the set of all subsets of OBS). So, $obs(t, i_{\text{SUT}})$ is a formal expression modelling the real test execution $\text{EXEC}(t, \text{SUT})$. In the context of an observational framework consisting of TEST, OBS, EXEC, and obs, the test assumption can be expressed in a more precise way:

$$\forall \text{SUT} \;\; \exists i_{\text{SUT}} \in \mathit{MOD} \;\; \forall t \in \mathit{TEST} : \;\; \text{EXEC}(t, \text{SUT}) \; = \; obs(t, i_{\text{SUT}}) \tag{1}$$

This could be paraphrased as follows: for all real SUTs that we are testing, it is assumed that there is a model i_{SUT}, such that if we would put the SUT and the model i_{SUT} in black boxes and would perform all possible experiments in TEST on both, then we would not be able to distinguish between the real SUT and the model i_{SUT}. Actually, this notion of testing is analogous to the ideas underlying testing equivalences [15,14].

In testing, a *verdict* is assigned based on the observations: $\nu_t : \mathcal{P}(\mathit{OBS}) \rightarrow \{\mathbf{fail}, \mathbf{pass}\}$, which allows to introduce the following abbreviation:

$$\text{SUT } \mathbf{passes}\ t \quad \Leftrightarrow_{\text{def}} \quad \nu_t(\text{EXEC}(t, \text{SUT})) = \mathbf{pass} \tag{2}$$

This is straightforwardly extended to a test suite $T \subseteq \mathit{TEST}$, and moreover a test suite fails if it does not pass:

$$\text{SUT } \mathbf{passes}\ T \quad \Leftrightarrow_{\text{def}} \quad \forall t \in T : \; \text{SUT } \mathbf{passes}\ t \tag{3}$$

$$\text{SUT } \mathbf{fails}\ T \quad \Leftrightarrow_{\text{def}} \quad \exists t \in T : \; \text{SUT } \not\mathbf{passes}\ t \tag{4}$$

Test Generation. The main gain of model-based testing is the systematic, algorithmic generation of test suites from a specification model for a given implementation relation: $gen_{\mathbf{imp}} : \mathit{SPEC} \rightarrow \mathcal{P}(\mathit{TEST})$,

The generated test cases should exactly detect those behaviours that are not correct with respect to the specification model and the implementation relation. A test suite is *sound* if all correct SUTs pass, i.e., there are no false alarms. The other way around, if no erroneous SUT passes a test suite, the test suite is called *exhaustive*, i.e., all possible failures are detected. For a specification s, a test suite T, and an implementation relation **imp**:

$$T \text{ is sound} \quad \Leftrightarrow_{\text{def}} \quad \forall i \in \mathit{MOD} : \; i \ \mathbf{imp}\ s \;\; \text{implies} \;\; i \ \mathbf{passes}\ T \tag{5}$$

$$T \text{ is exhaustive} \quad \Leftrightarrow_{\text{def}} \quad \forall i \in \mathit{MOD} : \; i \ \mathbf{imp}\ s \quad \text{if} \quad i \ \mathbf{passes}\ T \tag{6}$$

Soundness is minimal requirement for test suites. Exhaustive test suites do not exist in practice, because detecting all potential faults in an SUT would require an infinite number of infinitely long test cases: “Program testing can be used to show the presence of bugs, but never to show their absence!” [16]. Yet, for reasoning about model-based test generation algorithms, both soundness and exhaustiveness are important concepts. A theoretically exhaustive test generation algorithm will eventually detect all possible errors if the time of testing is unbounded. Practically, this means that every error has a non-zero probability of being detected, i.e., there are no errors that are fully undetectable.

Conformance Testing. Conformance testing involves assessing, by means of testing, whether an implementation conforms to its specification. Hence, the notions of conformance, expressed by **imp**, and of test execution, expressed by **passes**, have to be linked in such a way that test execution is a (semi-) decision procedure for conformance. This can indeed be achieved if soundness and exhaustiveness are proved on models:

$$\forall i \in \mathit{MOD}: \quad i \ \mathbf{imp}\ s \quad \text{iff} \quad \forall t \in T: \ \nu_t(\mathit{obs}(t,i)) = \mathbf{pass} \tag{7}$$

Once (7) has been shown it follows that

$$\begin{array}{ll}
 & \textsc{sut}\ \mathbf{passes}\ T \\
\text{iff} & (*\ \text{definition}\ \mathbf{passes}\ T\ \ *) \\
 & \forall t \in T:\ \textsc{sut}\ \mathbf{passes}\ t \\
\text{iff} & (*\ \text{definition}\ \mathbf{passes}\ t\ \ *) \\
 & \forall t \in T:\ \nu_t(\textsc{exec}(t,\textsc{sut})) = \mathbf{pass} \\
\text{iff} & (*\ \text{test assumption (1)}\ \ *) \\
 & \forall t \in T:\ \nu_t(\mathit{obs}(t, i_{\textsc{sut}})) = \mathbf{pass} \\
\text{iff} & (*\ \text{soundness and exhaustiveness on models (7)}\ \ *) \\
 & i_{\textsc{sut}}\ \mathbf{imp}\ s \\
\text{iff} & \textsc{sut}\ \text{conforms to}\ s
\end{array}$$

Thus, testing is indeed a decision procedure for **imp**-conformance if the test assumption (1) and (7) hold. In case of test generation $\mathit{gen}_{\mathbf{imp}} : \mathit{SPEC} \rightarrow \mathcal{P}(\mathit{TEST})$ the proof obligation shall hold for any $s \in \mathit{SPEC}$:

$$\forall s \in \mathit{SPEC}\ \ \forall i \in \mathit{MOD}: \quad i\ \mathbf{imp}\ s \ \ \text{iff} \ \ i\ \mathbf{passes}\ \mathit{gen}_{\mathbf{imp}}(s) \tag{8}$$

Test Selection. A sound and exhaustive test generation algorithm can generate many more test cases than can ever be executed. Even testing the addition of two 32-bit integers, which could easily be automated by writing a test generation algorithm that enumerates all $2^{32} \times 2^{32} = 1.8\ 10^{19}$ possible test cases, would require 584,542 years of test execution if one test case would take 1 μsec.

Practical test generation algorithms use *test selection criteria* to generate a feasible and executable selection of sound, but not exhaustive test cases. The aim is to select test cases in such a way that they provide a high chance of detecting failures, and give confidence that an SUT that passes is indeed conforming, within given constraints of testing time and effort.

Selection criteria, also referred to as test adequacy criteria, are based on heuristics, experience, gut feeling, and expert domain knowledge, so human influence is prominent. General selection criteria for model-based testing express when a model is considered sufficiently *covered* by test cases. Examples are state- and transition coverage for state-based models, and condition coverage for guarded-command specifications. Selection criteria may also be specific for a particular model or domain, such as a domain expert having knowledge about particular critical behaviours.

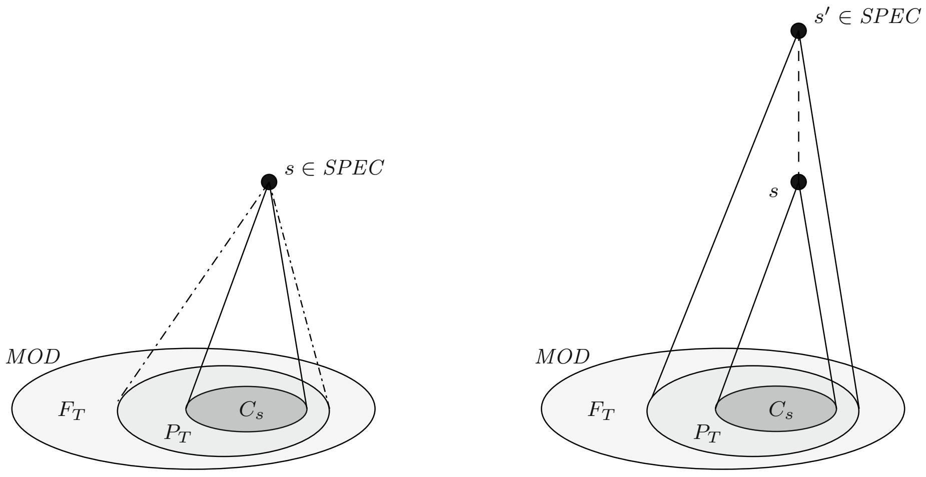


Fig. 2. Test selection as a subset of *MOD*

Test selection is an important yet difficult topic, with various approaches, see, for example, the use of test purposes [26,48], the use of metrics [13,18], approximate analysis [27], and coverage analysis [22]. We elaborate here a bit on an approach in which test selection is considered as a measure-theoretic question on the domain of implementations *MOD* [10,9]; this approach will be reconsidered in the context of learning in Sect. 4.

In Fig. 2, left-hand side, *MOD* is represented together with a specification $s \in SPEC$. The specification s partitions *MOD* into conforming implementations $C_s = \{m \mid m \ \mathbf{imp}\ s\}$ and nonconforming ones $MOD \backslash C_s$. A test suite $T \subseteq TEST$ partitions *MOD* into passing implementations $P_T = \{m \mid m \ \mathbf{passes}\ T\}$ and failing ones $F_T = MOD \backslash P_T$. Ideally, the test suite T for s is sound and exhaustive so that C_s and P_T coincide and all and only nonconforming implementations are detected by T. In practice, however, test suites are only sound, $C_s \subsetneq P_T$, so that they detect only but not all nonconforming implementations. There is an area of nonconforming yet passing implementations $P_T \backslash C_s$. Test selection aims at minimizing this area, or equivalently, optimizing the area F_T. The area F_T is a measure for the level of exhaustiveness of a sound test suite.

Suppose we have a monotonic, with $\subseteq$-increasing measure on *MOD*: $\mu : \mathcal{P}(MOD) \rightarrow \mathbb{R}_{\geq 0}$, then the coverage of a test suite T is expressed by

$$\frac{\mu(F_T)}{\mu(MOD \backslash C_s)}$$

i.e., the value of detected erroneous implementations normalized with respect to all erroneous implementations. If there is also a function $cost : \mathcal{P}(TEST) \rightarrow \mathbb{R}_{\geq 0}$ expressing the cost and effort of executing test suite T, then test selection is the optimization problem of choosing T such that the coverage is maximized and the cost is minimized. In practice, this will often amount to optimizing the coverage within given constraints on cost.

Another way of looking at the above view on test selection is depicted in the right-hand side of Fig. 2. The selected test suite T is a sound and exhaustive test suite for some other, weaker specification s', i.e., $P_T = C_{s'}$. Test selection can be cast into the specification domain as a transformation of s into s'. A sound and exhaustive test suite is then generated from s', which is a weaker specification in the sense that s' allows more conforming implementations. Suppose we can define a distance function on specifications then exhaustiveness and test selection can be quantified in the specification domain, and corresponds to minimizing the distance between s and s' constraint by maximum admissible test cost [18].

Conclusion. For reasoning about formal, model-based testing we need a formal specification language *SPEC*, a domain of models of implementations *MOD*, an implementation relation $\textbf{imp} \subseteq \mathit{MOD} \times \mathit{SPEC}$ expressing correctness, a domain of test cases *TEST*, a test execution procedure $\textbf{passes} \subseteq \mathit{MOD} \times \mathit{TEST}$ expressing when a model of an implementation passes a test case, a test generation algorithm $\mathit{gen}_{\textbf{imp}} : \mathit{SPEC} \rightarrow \mathcal{P}(\mathit{TEST})$, a proof that a model of an implementation passes a generated test suite if and only if it is **imp**-correct, and the test assumption that any SUT can be modelled by a model in *MOD*. Then model-based testing is a decision procedure for **imp**-conformance. Yet, practical test suites are sound but not exhaustive so that test selection is necessary. Test selection quantifies the level of exhaustiveness by test coverage, and then optimizes test coverage against test cost.

The next section will elaborate most of these concepts for the formalism of labelled transition systems and the **ioco**-implementation relation. This means that we will use (variants of) labelled transition systems for *SPEC*, *MOD*, and *TEST*, that conformance is expressed as the relation **ioco** on labelled transition systems, that test execution of a labelled transition system with an implementation is defined, and that a test generation algorithm is presented that is proved to generate sound and exhaustive test suites.

Also other elaborations for other kinds of formal models are possible, e.g., Finite State Machines (FSM, Mealy Machines) [38], Abstract Data Types [20], object oriented formalisms [12], or (mathematical) functions [29].

In Sect. 4 we will use the formalizations in this section to relate model-based testing to test-based modelling, also called learning.

3 Model-Based Testing for Labelled Transition Systems

This section presents an overview of the formal test theory for labelled transition systems using the **ioco**-conformance relation; see [41,43] for a more elaborate treatment.

Models. In the **ioco**-test theory, specification models, implementations, and test cases are all expressed as labelled transition systems.

Definition 1. *A* labelled transition system with inputs and outputs *is a 5-tuple* $\langle Q, L_I, L_U, T, q_0\rangle$ *where* Q *is a countable, non-empty set of* states*;* L_I *is a* *countable set of* input labels*; L_U is a countable set of* output labels*, such that $L_I \cap L_U = \emptyset$; $T \subseteq Q \times (L_I \cup L_U \cup \{\tau\}) \times Q$, with $\tau \notin L_I \cup L_U$, is the* transition relation*; and $q_0 \in Q$ is the* initial state*.*

The labels in L_I and L_U represent the inputs and outputs, respectively, of a system, i.e., the system's possible interactions with its environment. (The 'U' refers to 'uitvoer', the Dutch word for 'output', which is preferred for historical reasons, and to avoid confusion between L_O (letter 'O') and L_0 (digit zero)). Inputs are usually decorated with '?' and outputs with '!'. We use $L = L_I \cup L_U$ when we abstract from the distinction between inputs and outputs.

The execution of an action is modelled as a transition: $(q, \mu, q') \in T$ expresses that the system, when in state q, may perform action μ, and go to state q' . This is more elegantly denoted as $q \xrightarrow{\mu} q'$. Transitions can be composed: $q \xrightarrow{\mu} q' \xrightarrow{\mu'} q''$, which is written as $q \xrightarrow{\mu \cdot \mu'} q''$.

Internal transitions are labelled by the special action τ ($\tau \notin L$), which is assumed to be unobservable for the system's environment. Consequently, the observable behaviour of a system is captured by the system's ability to perform sequences of observable actions. Such a sequence of observable actions, say σ, is obtained from a sequence of actions under abstraction from the internal action τ, and it is denoted by $\overset{\sigma}{\Longrightarrow}$. If, for example, $q \xrightarrow{a \cdot \tau \cdot \tau \cdot b \cdot c \cdot \tau} q'$ $(a, b, c \in L)$, then we write $q \overset{a \cdot b \cdot c}{\Longrightarrow} q'$ for the τ-abstracted sequence of observable actions. We say that q is able to perform the *trace* $a \cdot b \cdot c \in L^*$. Here, the set of all finite sequences over L is denoted by L^*, with ϵ denoting the empty sequence. If $\sigma_1, \sigma_2 \in L^*$ are finite sequences, then $\sigma_1 \cdot \sigma_2$ is the concatenation of σ_1 and σ_2. Some more, standard notations and definitions are given in Definitions 2 and 3.

Definition 2. *Let $p = \langle Q, L_I, L_U, T, q_0 \rangle$ be a labelled transition system with $q, q' \in Q$, $\mu, \mu_i \in L \cup \{\tau\}$, $a, a_i \in L$, and $\sigma \in L^*$.*

$$
\begin{array}{lll}
q \xrightarrow{\mu} q' & \Leftrightarrow_{\text{def}} & (q, \mu, q') \in T \\
q \xrightarrow{\mu_1 \cdot \ldots \cdot \mu_n} q' & \Leftrightarrow_{\text{def}} & \exists q_0, \ldots, q_n : \ q = q_0 \xrightarrow{\mu_1} q_1 \xrightarrow{\mu_2} \ldots \xrightarrow{\mu_n} q_n = q' \\
q \xrightarrow{\mu_1 \cdot \ldots \cdot \mu_n} & \Leftrightarrow_{\text{def}} & \exists q' : \ q \xrightarrow{\mu_1 \cdot \ldots \cdot \mu_n} q' \\
q \not\xrightarrow{\mu_1 \cdot \ldots \cdot \mu_n} & \Leftrightarrow_{\text{def}} & \textit{not } \exists q' : \ q \xrightarrow{\mu_1 \cdot \ldots \cdot \mu_n} q' \\
q \overset{\epsilon}{\Longrightarrow} q' & \Leftrightarrow_{\text{def}} & q = q' \ \text{ or } \ q \xrightarrow{\tau \cdot \ldots \cdot \tau} q' \\
q \overset{a}{\Longrightarrow} q' & \Leftrightarrow_{\text{def}} & \exists q_1, q_2 : \ q \overset{\epsilon}{\Longrightarrow} q_1 \xrightarrow{a} q_2 \overset{\epsilon}{\Longrightarrow} q' \\
q \overset{a_1 \cdot \ldots \cdot a_n}{\Longrightarrow} q' & \Leftrightarrow_{\text{def}} & \exists q_0 \ldots q_n : \ q = q_0 \overset{a_1}{\Longrightarrow} q_1 \overset{a_2}{\Longrightarrow} \ldots \overset{a_n}{\Longrightarrow} q_n = q' \\
q \overset{\sigma}{\Longrightarrow} & \Leftrightarrow_{\text{def}} & \exists q' : \ q \overset{\sigma}{\Longrightarrow} q' \\
q \overset{\sigma}{\not\Longrightarrow} & \Leftrightarrow_{\text{def}} & \textit{not } \exists q' : q \overset{\sigma}{\Longrightarrow} q'
\end{array}
$$

In our reasoning about labelled transition systems we will not always distinguish between a transition system and its initial state. If $p = \langle Q, L_I, L_U, T, q_0 \rangle$, we will identify the process p with its initial state q_0, and, e.g., we write $p \overset{\sigma}{\Longrightarrow}$ instead of $q_0 \overset{\sigma}{\Longrightarrow}$.

Definition 3. *Let p be a (state of a) labelled transition system, P a set of states, $A \subseteq L$ a set of labels, and $\sigma \in L^*$.*

1. $traces(p) =_{\rm def} \{ \sigma \in L^* \mid p \overset{\sigma}{\Longrightarrow} \}$
2. $p\,\textbf{after}\,\sigma =_{\rm def} \{ p' \mid p \overset{\sigma}{\Longrightarrow} p' \}$
3. $P\,\textbf{after}\,\sigma =_{\rm def} \bigcup \{ p\,\textbf{after}\,\sigma \mid p \in P \}$
4. $P\,\textbf{refuses}\,A =_{\rm def} \exists p \in P,\ \forall \mu \in A \cup \{\tau\} :\ p \overset{\mu}{\not\longrightarrow}$

The class of labelled transition systems with inputs in L_I and outputs in L_U is denoted as $\mathcal{LTS}(L_I, L_U)$. For technical reasons we restrict this class to *strongly converging* and *image finite* systems. Strong convergence means that infinite sequences of τ-actions are not allowed to occur. Image finiteness means that the number of non-deterministically reachable states shall be finite, i.e., for any σ, $p\,\textbf{after}\,\sigma$ shall be finite.

Representing Labelled Transition Systems. To represent labelled transition systems we use either graphs (as in Fig. 3), or expressions in a process-algebraic-like language with the following syntax:

$$B \ ::= \ a\,;B \ \mid \ \mathbf{i}\,;B \ \mid \ \mathbf{\Sigma}\,\mathcal{B} \ \mid \ B\,|[\,G\,]|\,B \ \mid \ P$$

Expressions in this language are called behaviour expressions, and they define labelled transition systems following the axioms and rules given in Table 1.

Table 1. Structural operational semantics

$$\frac{}{a\,;B \xrightarrow{a} B} \qquad \frac{}{\mathbf{i}\,;B \xrightarrow{\tau} B} \qquad \frac{B \xrightarrow{\mu} B'}{\Sigma\,\mathcal{B} \xrightarrow{\mu} B'}\ B \in \mathcal{B},\ \mu \in L \cup \{\tau\}$$

$$\frac{B_1 \xrightarrow{\mu} B_1'}{B_1\,|[G]|\,B_2 \xrightarrow{\mu} B_1'\,|[G]|\,B_2} \qquad \frac{B_2 \xrightarrow{\mu} B_2'}{B_1\,|[G]|\,B_2 \xrightarrow{\mu} B_1\,|[G]|\,B_2'} \qquad \mu \in (L \cup \{\tau\}) \backslash G$$

$$\frac{B_1 \xrightarrow{a} B_1',\ B_2 \xrightarrow{a} B_2'}{B_1\,|[G]|\,B_2 \xrightarrow{a} B_1'\,|[G]|\,B_2'}\ a \in G \qquad \frac{B_P \xrightarrow{\mu} B'}{P \xrightarrow{\mu} B'}\ P := B_P,\ \mu \in L \cup \{\tau\}$$

In that table, $a \in L$ is a label, B is a behaviour expression, $\mathcal{B}$ is a countable set of behaviour expressions, $G \subseteq L$ is a set of labels, and P is a *process name*, which must be linked to a named behaviour expression by a process definition of the form $P := B_P$. In addition, we use $B_1 \,\Box\, B_2$ as an abbreviation for $\mathbf{\Sigma}\{B_1, B_2\}$, **stop** to denote $\mathbf{\Sigma}\,\emptyset$, $||$ as an abbreviation for $|[\,L\,]|$, i.e., synchronization on all observable actions, and $|||$ as an abbreviation for $|[\,\emptyset\,]|$, i.e., full interleaving without synchronization.

Input-Output Transition Systems. In the **ioco**-test theory a specification is a labelled transition system in $\mathcal{LTS}(L_I, L_U)$. In order to formally reason about an SUT the assumption is made that the SUT behaves as if it were some kind of behavioural, formal model. This assumption is referred to as the test assumption or test hypothesis, and this model is called an implementation; see Sect. 2. In the **ioco**-test theory the test assumption is that an SUT behaves as if it were a labelled transition system that is always able to perform any input action, i.e., all inputs are enabled in all states. Such a system is defined as an *input-output transition system.* The class of such input-output transition systems is denoted by $\mathcal{IOTS}(L_I, L_U) \subseteq \mathcal{LTS}(L_I, L_U)$.

Definition 4. *An* input-output transition system *is a labelled transition system with inputs and outputs* $\langle Q, L_I, L_U, T, q_0 \rangle$ *where all input actions are enabled in any reachable state:* $\forall \sigma, q : \; q_0 \overset{\sigma}{\Longrightarrow} q$ implies $\forall a \in L_I : \; q \overset{a}{\Longrightarrow}$

A state of a system where no outputs or internal actions are enabled, and consequently the system is forced to wait until its environment provides an input, is called *suspended*, or *quiescent.* An observer looking at a quiescent system does not see any outputs. This particular observation of seeing nothing can itself be considered as an event, which is denoted by δ $(\delta \notin L \cup \{\tau\})$; $p \xrightarrow{\delta} p$ expresses that p allows the observation of quiescence. Also these transitions can be composed, e.g., $p \xRightarrow{\delta \cdot ?a \cdot \delta \cdot ?b \cdot !x}$ expresses that initially p is quiescent, i.e., does not produce outputs, but p does accept input action $?a$, after which there are again no outputs; when then input $?b$ is performed, the output $!x$ is produced. We use L_δ for $L \cup \{\delta\}$, and traces that may contain the quiescence action δ are called *suspension traces.*

Definition 5. *Let* $p = \langle Q, L_I, L_U, T, q_0 \rangle \in \mathcal{LTS}(L_I, L_U)$.

1. *A state* q *of* p *is* quiescent, *denoted by* $\delta(q)$, *if* $\forall \mu \in L_U \cup \{\tau\} : \; q \overset{\mu}{\not\longrightarrow}$
2. $p_\delta \; =_{\text{def}} \; \langle \, Q, \; L_I, \; L_U \cup \{\delta\}, \; T \cup T_\delta, \; q_0 \, \rangle$,
   with $T_\delta \; =_{\text{def}} \; \{ \, q \xrightarrow{\delta} q \mid q \in Q, \; \delta(q) \, \}$
3. *The* suspension traces *of* p *are* $\mathit{Straces}(p) \; =_{\text{def}} \; \{ \, \sigma \in L_\delta^* \mid p_\delta \overset{\sigma}{\Longrightarrow} \, \}$

From now on we will usually include δ-transitions in the transition relations, i.e., we consider p_δ instead of p, unless otherwise indicated. Definitions 2 and 3 also apply to transition systems with label set L_δ.

The Implementation Relation ioco. An implementation relation is intended to precisely define when an implementation is correct with respect to a specification. We use the implementation relation **ioco**, which is abbreviated from input-output conformance. Informally, an implementation $i \in \mathcal{IOTS}(L_I, L_U)$ is **ioco**-conforming to specification $s \in \mathcal{LTS}(L_I, L_U)$ if any experiment derived from s and executed on i leads to an output (including quiescence) from i that is foreseen by s. We define **ioco** as a special case of the more general class of relations $\mathbf{ioco}_\mathcal{F}$, where $\mathcal{F} \subseteq L_\delta^*$ is a set of suspension traces, which typically depends on the specification s.

Definition 6. *Let* q *be a state in a transition system,* Q *be a set of states,* $i \in \mathcal{IOTS}(L_I, L_U)$, $s \in \mathcal{LTS}(L_I, L_U)$, *and* $\mathcal{F} \subseteq L_\delta^*$, *then*

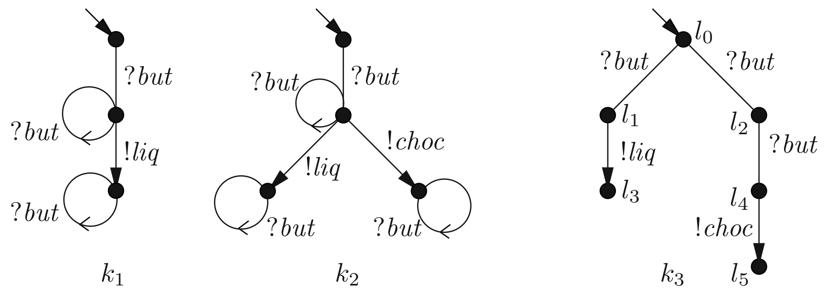


Fig. 3. Example labelled transition systems

1. $out(q) =_{\text{def}} \{\, x \in L_U \mid q \xrightarrow{x} \,\} \cup \{\, \delta \mid \delta(q) \,\}$
2. $out(Q) =_{\text{def}} \bigcup \{\, out(q) \mid q \in Q \,\}$
3. $i\ \mathbf{ioco}_{\mathcal{F}}\ s \quad \Leftrightarrow_{\text{def}} \quad \forall \sigma \in \mathcal{F} : \quad out(\, i\ \mathbf{after}\ \sigma\,) \ \subseteq\ out(\, s\ \mathbf{after}\ \sigma\,)$
4. $i\ \mathbf{ioco}\ s \quad \Leftrightarrow_{\text{def}} \quad i\ \mathbf{ioco}_{Straces(s)}\ s$

Example 1. Figure 3 presents three examples of labelled transition systems modelling candy machines. There is an input action for pushing a button $?but$, and there are outputs for obtaining chocolate $!choc$ and liquorice $!liq$: $L_I = \{?but\}$ and $L_U = \{!liq, !choc\}$.

Since $k_1, k_2 \in \mathcal{IOTS}(L_I, L_U)$ they can be both specifications and implementations; k_3 is not input-enabled, and can only be a specification. We have that $out(\, k_1\ \mathbf{after}\ ?but\,) = \{!liq\} \subseteq \{!liq, !choc\} = out(\, k_2\ \mathbf{after}\ ?but\,)$; so we get now k_1 **ioco** k_2, but $k_2\ \mathbf{io\not{c}o}\ k_1$. For k_3 we have $out(\, k_3\ \mathbf{after}\ ?but\,) = \{!liq, \delta\}$ since $\delta(l_2)$, and $out(\, k_3\ \mathbf{after}\ ?but\cdot?but\,) = \{!choc\}$, so both $k_1, k_2\ \mathbf{io\not{c}o}\ k_3$.

The importance of having suspension actions δ in the set $\mathcal{F}$ over which **ioco** quantifies is illustrated in Fig. 4. It holds that $out(\, r_1\ \mathbf{after}\ ?but\cdot?but\,) = out(\, r_2\ \mathbf{after}\ ?but\cdot?but\,) = \{!liq, !choc\}$, but we have $out(\, r_1\ \mathbf{after}\ ?but\cdot\delta\cdot?but\,) = \{!liq, !choc\} \supset \{!choc\} = out(\, r_2\ \mathbf{after}\ ?but\cdot\delta\cdot?but\,)$. So, without δ in these traces r_1 and r_2 would be considered implementations of each other in both directions, whereas with δ, r_2 **ioco** r_1 but $r_1\ \mathbf{io\not{c}o}\ r_2$.

Underspecification and the Implementation Relation uioco. The implementation relation **ioco** allows to have partial specifications. A partial specification does not specify the required behaviour of the implementation after all possible traces. This corresponds to the fact that specifications may be non-input enabled, and inclusion of *out*-sets is only required for suspension traces that explicitly occur in the specification. Traces that do not explicitly occur are called underspecified. There are different ways of dealing with underspecified traces. The relation **uioco** does it in a slightly different manner than **ioco**. For the rationale consider Example 2.

Example 2. Consider k_3 of Fig. 3 as a specification. Since k_3 is not input-enabled, it is a partial specification. For example, $?but\cdot?but\cdot?but$ is an underspecified trace, and any implementation behaviour is allowed after it. On the other hand, $?but$ is clearly specified; the allowed outputs after it are $!liq$ and δ. For the trace $?but{\cdot}?but$ the situation is less clear. According to **ioco** the expected output after $?but{\cdot}?but$ is $out(\,k_3\,\textbf{after}\,?but{\cdot}?but\,) = \{!choc\}$. But suppose that in the first $?but$-transition k_3 moves nondeterministically to state l_1 (the left branch) then one might argue that the second $?but$-transition is underspecified, and that, consequently, any possible behaviour is allowed in an implementation. This is exactly where **ioco** and **uioco** differ: **ioco** postulates that $?but{\cdot}?but$ is not an underspecified trace, because there exists a state where it is specified, whereas **uioco** states that $?but{\cdot}?but$ is underspecified, because there exists a state where it is underspecified.

Formally, **ioco** quantifies over $\mathcal{F} = Straces(s)$, which are all possible suspension traces of the specification s. The relation **uioco** quantifies over $\mathcal{F} = Utraces(s) \subseteq Straces(s)$, which are the suspension traces without the possibly underspecified traces, i.e., all suspension traces σ of s for which it is *not* possible that a prefix σ_1 of σ ($\sigma = \sigma_1{\cdot}a{\cdot}\sigma_2$) leads to a state of s where the remainder $a{\cdot}\sigma_2$ of σ is underspecified, that is, a is refused.

Definition 7. *Let $i \in \mathcal{IOTS}(L_I, L_U)$, and $s \in \mathcal{LTS}(L_I, L_U)$.*

1. $Utraces(s) \;=_{\rm def}\; \{\ \sigma \in Straces(s)\ \mid\ \forall \sigma_1, \sigma_2 \in L_\delta^*,\ a \in L_I :$
   $\qquad \sigma = \sigma_1{\cdot}a{\cdot}\sigma_2 \text{ implies not } s\,\textbf{after}\,\sigma_1\,\textbf{refuses}\,\{a\}\ \}$
2. $i\ \textbf{uioco}\ s \quad \Leftrightarrow_{\rm def} \quad i\ \textbf{ioco}_{Utraces(s)}\ s$

Example 3. Because $Utraces(s) \subseteq Straces(s)$ it is evident that **uioco** is not stronger than **ioco**. That it is strictly weaker follows from the following example. Take k_3 in Fig. 3 as a (partial) specification, and consider r_1 and r_2 from Fig. 4 as implementations. Then r_2 $\cancel{\textbf{ioco}}$ k_3 because $!liq \in out(r_2\,\textbf{after}\,?but{\cdot}?but\,)$ and $!liq \notin out(\,k_3\,\textbf{after}\,?but{\cdot}?but\,)$. But r_2 **uioco** k_3 because we have $?but{\cdot}?but \notin Utraces(k_3)$. Also r_1 $\cancel{\textbf{ioco}}$ k_3, but in this case also r_1 $\cancel{\textbf{uioco}}$ k_3. The reason for this is that we have $?but{\cdot}\delta{\cdot}?but \in Utraces(k_3)$, $!liq \in out(\,r_1\,\textbf{after}\,?but{\cdot}\delta{\cdot}?but\,)$ and $!liq \notin out(\,k_3\,\textbf{after}\,?but{\cdot}\delta{\cdot}?but\,)$.

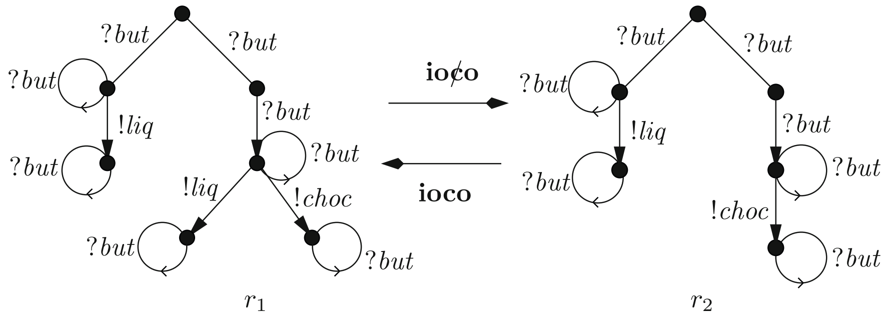


Fig. 4. More labelled transition systems

Test Cases. For the generation of test cases from labelled transition system specifications, which can test SUTs that behave as input-output transition systems, we must first define what test cases are. Then we discuss what test execution is, and what it means to pass a test. A test generation algorithm is given, and soundness and exhaustiveness are discussed.

A test case is a specification of the behaviour of a tester in an experiment carried out on an SUT. The behaviour of such a tester is also modelled as a special kind of input-output transition system, but, naturally, with inputs and outputs exchanged. Consequently, input-enabledness of a test case means that all actions in L_U (i.e., the set of outputs of the implementation) are enabled. For observing quiescence we add a special label θ to the transition systems modelling tests ($\theta \notin L$).

Definition 8. *A* test case *t for an implementation with inputs L_I and outputs L_U is an input-output transition system $\langle Q, L_U, L_I \cup \{\theta\}, T, q_0\rangle \in \mathcal{IOTS}(L_U, L_I \cup \{\theta\})$ generated following the next fragment of the syntax for behaviour expressions, where* **pass** *and* **fail** *are process names:*

$$\begin{array}{rll} t ::= & \textbf{pass} & \\ | & \textbf{fail} & \\ | & \Sigma\ \{\ x\ ;\ t \mid x \in L_U \cup \{a\}\ \} & \textit{for some } a \in L_I \\ | & \Sigma\ \{\ x\ ;\ t \mid x \in L_U \cup \{\theta\}\ \} & \\ & \textit{where}\quad \textbf{pass} := \Sigma\ \{\ x\,;\, \textbf{pass}\ \mid\ x \in L_U \cup \{\theta\}\ \} & \\ & \qquad\qquad \textbf{fail}\ := \Sigma\ \{\ x\,;\, \textbf{fail}\ \mid\ x \in L_U \cup \{\theta\}\ \} & \end{array}$$

The class of test cases for implementations with inputs L_I and outputs L_U is denoted as $\mathcal{TTS}(L_U, L_I)$. A set of test cases is called a *test suite* $T \subseteq \mathcal{TTS}(L_U, L_I)$.

Test Execution. Test cases are run by putting them in parallel with the implementation, where inputs of the test case synchronize with the outputs of the implementation, and vice versa. Basically, this can be modelled using the behaviour-expression operator $\|$. Since, however, we added the special label θ to test cases to test for quiescence, this operator has to be extended a bit, and is then denoted as $\rceil\!|$.

Because of nondeterminism in implementations, it may be the case that testing the same implementation with the same test case leads to different test runs. Test execution consists of performing all possible test runs. Each test run leads to an observation which is the trace executed until a **pass**- or **fail**-state is reached. Thus, test execution leads to a set of observations. An implementation passes a test case if and only if all its test runs lead to a pass verdict of the test case. All this is reflected in the following definition.

Definition 9. *Let $t \in \mathcal{TTS}(L_U, L_I)$ and $i \in \mathcal{IOTS}(L_I, L_U)$.*

1. Running *a test case t with an implementation i is expressed by the parallel operator $\rceil\!|\ : \mathcal{TTS}(L_U, L_I) \times \mathcal{IOTS}(L_I, L_U) \rightarrow \mathcal{LTS}(L_I \cup L_U \cup \{\delta\})$ which*

is defined by the following inference rules:

$$\frac{i \xrightarrow{\tau} i'}{t \rceil\!| \, i \xrightarrow{\tau} t \rceil\!| \, i'} \qquad \frac{t \xrightarrow{a} t', \ i \xrightarrow{a} i'}{t \rceil\!| \, i \xrightarrow{a} t' \rceil\!| \, i'} \ a \in L_I \cup L_U \qquad \frac{t \xrightarrow{\theta} t', \ i \xrightarrow{\delta}}{t \rceil\!| \, i \xrightarrow{\delta} t' \rceil\!| \, i}$$

2. *An* observation *of a test run of t with i is a trace of $t \rceil\!| \, i$ leading to one of the states* **pass** *or* **fail** *of t:*

$$obs(t,i) \ =_{\text{def}} \ \{ \ \sigma \in L_\delta^* \ \mid \ \exists i' : \ t \rceil\!| \, i \overset{\sigma}{\Longrightarrow} \mathbf{pass} \rceil\!| \, i' \ \text{ or } \ t \rceil\!| \, i \overset{\sigma}{\Longrightarrow} \mathbf{fail} \rceil\!| \, i' \ \}$$

3. *Implementation i* passes *test case t if all observations go to the* **pass***-state of t:*

$$i \ \mathbf{passes} \ t \quad \Leftrightarrow_{\text{def}} \quad \forall \sigma \in obs(t,i) \ \ \exists i' : \ \ t \rceil\!| \, i \overset{\sigma}{\Longrightarrow} \mathbf{pass} \rceil\!| \, i'$$

4. *An implementation i* passes *a test suite T if it passes all test cases in T:*

$$i \ \mathbf{passes} \ T \quad \Leftrightarrow_{\text{def}} \quad \forall t \in T : \ i \ \mathbf{passes} \ t$$

If i does not pass a test case or a test suite, it **fails***.*

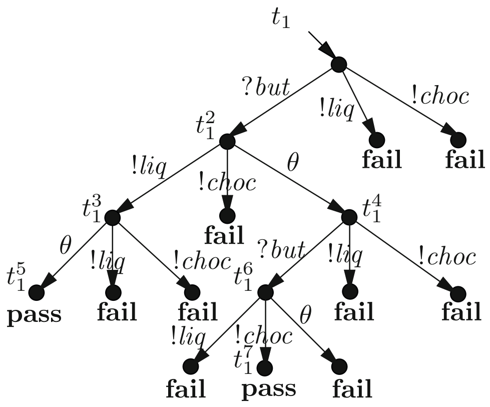


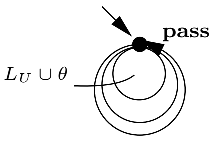
Fig. 5. A test case

Test Generation. Now all ingredients are there to present an algorithm to generate test cases from a labelled transition system specification, which test implementations for **ioco**-correctness.

Algorithm 1. *Let $s \in \mathcal{LTS}(L_I, L_U)$ be a specification, and let S be a set of states with initially $S = s$ **after** ϵ.*

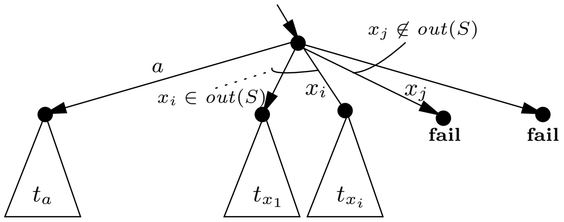
A test case $t \in \mathcal{TTS}(L_U, L_I)$ is obtained from a non-empty set of states S by a finite number of recursive applications of one of the following three nondeterministic choices:

1.



$t := \mathbf{pass}$

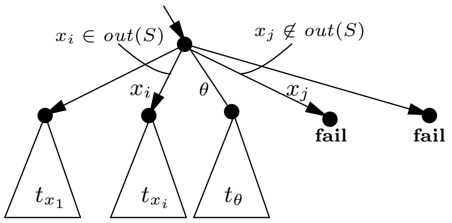
2.



$$\begin{array}{rl} t := & a \;;\; t_a \\ & \Box\; \Sigma\; \{\; x_j \;;\; \mathbf{fail} \mid x_j \in L_U,\; x_j \notin out(S) \;\} \\ & \Box\; \Sigma\; \{\; x_i \;;\; t_{x_i} \mid x_i \in L_U,\; x_i \in out(S) \;\} \end{array}$$

where $a \in L_I$ *such that* $S\,\mathbf{after}\,a \neq \emptyset$, t_a *is obtained by recursively applying the algorithm for the set of states* $S\,\mathbf{after}\,a$, *and for each* $x_i \in out(S)$, t_{x_i} *is obtained by recursively applying the algorithm for the set of states* $S\,\mathbf{after}\,x_i$ *.*

3.



$$\begin{array}{rl} t := & \Sigma\; \{\; x_j \;;\; \mathbf{fail} \mid x_j \in L_U,\; x_j \notin out(S) \;\} \\ & \Box\; \Sigma\; \{\; \theta \;;\; \mathbf{fail} \mid \delta \notin out(S) \;\} \\ & \Box\; \Sigma\; \{\; x_i \;;\; t_{x_i} \mid x_i \in L_U,\; x_i \in out(S) \;\} \\ & \Box\; \Sigma\; \{\; \theta \;;\; t_\theta \mid \delta \in out(S) \;\} \end{array}$$

where for each $x_i \in out(S)$, t_{x_i} *is obtained by recursively applying the algorithm for the set of states* $S\,\mathbf{after}\,x_i$, *and* t_θ *is obtained by recursively applying the algorithm for the set of states* $S\,\mathbf{after}\,\delta$ *.*

Algorithm 1 generates a test case from a set of states S. This set represents the set of all possible states in which the specification can be at the given stage of the test case generation. Initially $S = s \textbf{ after } \epsilon = q_0 \textbf{ after } \epsilon$, so that the first transition of the test case is derived from the initial state(s) of the specification. Then the remaining part of the test case is recursively derived from the specification states reachable from the initial states via this first test case transition.

The algorithm is nondeterministic in the sense that in each recursive step it can be continued in three different ways. Each choice results in another, valid test case:

choice 1: The test case can be terminated by ending the recursion with the single-state test case **pass**, which is always a sound test case.

choice 2: The test case can continue with supplying an input $a \in L_I$ allowed by the specification ($S \textbf{ after } a \neq \emptyset$). After action a the test case continues as t_a, which is obtained by recursive application of the algorithm with the set of states $S \textbf{ after } a$. Moreover, t is prepared to accept any output of the SUT (not quiescence) that might occur before a is supplied. Analogous to t_a, each t_{x_i} is obtained from $S \textbf{ after } x_i$.

choice 3: The test case can wait for an output of the SUT and check it, or conclude that the SUT is quiescent, i.e., produces 'output' δ. If the output, whether real or quiescence, is not allowed, i.e., $x_j \not\in out(S)$, the test case terminates with **fail**. If the response is allowed the algorithm continues with recursively generating a test case from the set of states $S \textbf{ after } x_i$.

Example 4. Test case t_1 of Figure 5 can be generated from k_3 in Figure 3 with Algorithm 1:

1. Initially, $S := k_3 \textbf{ after } \epsilon = \{l_0\}$.
2. In the first step input $?but$ is tried: $t_1 := ?but; t_1^2 \;\Box\; !liq; \textbf{fail} \;\Box\; !choc; \textbf{fail}$, after which $S := \{l_0\} \textbf{ after } ?but = \{l_1, l_2\}$.
3. The allowed outputs are checked: $out(S) = out(\{l_1, l_2\}) = \{!liq, \delta\}$. This leads to the test case $t_1^2 := !liq; t_1^3 \;\Box\; !choc; \textbf{fail} \;\Box\; \theta; t_1^4$.
4. For t_1^3 we continue with $S := \{l_1, l_2\} \textbf{ after } !liq = \{l_3\}$ for which it is checked that no output occurs: $t_1^3 := !liq; \textbf{fail} \;\Box\; !choc; \textbf{fail} \;\Box\; \theta; t_1^5$.
5. The test case is stopped: $t_1^5 := \textbf{pass}$.
6. Further with t_1^4: this is the test case after quiescence has been observed; t_1^4 is generated from $S := \{l_1, l_2\} \textbf{ after } \delta = \{l_2\}$. From $\{l_2\}$ another input $?but$ can be supplied: $t_1^4 := ?but; t_1^6 \;\Box\; !liq; \textbf{fail} \;\Box\; !choc; \textbf{fail}$.
7. Now output is checked: $t_1^6 := !liq; \textbf{fail} \;\Box\; !choc; t_1^7 \;\Box\; \theta; \textbf{fail}$.
8. After this the test case is stopped: $t_1^7 := \textbf{pass}$.

Soundness and Exhaustiveness. Algorithm 1 is correct, in the sense that the generated test suites are able to detect all, and only all, non-**ioco** correct implementations. This is expressed by the properties of soundness and exhaustiveness; see Sect. 2. This means that testing for **ioco** according to Algorithm 1 and Def. 9 is indeed a decision procedure for **ioco**-correctness. Of course, exhaustiveness is merely a theoretical property: for realistic systems exhaustive test suites would be infinite. But yet, exhaustiveness does express that there are no **ioco**-errors that are undetectable.

Theorem 2. *Algorithm 1 generates sound test cases, and the set of all test cases that can be generated with it is exhaustive for* **ioco** *and* s.

Example 5. In Example 4 test case t_1 of Figure 5 was generated from specification k_3 in Figure 3. For t_1 with k_1 there is one test run:
$t_1 \rceil| \, k_1 \xRightarrow{?but \cdot !liq \cdot \delta} \mathbf{pass} \rceil| \, k_1'$, so k_1 **passes** t_1.
For t_1 with k_2 there are two test runs:
$t_1 \rceil| \, k_2 \xRightarrow{?but \cdot !liq \cdot \delta} \mathbf{pass} \rceil| \, k_2'$, and $t_1 \rceil| \, k_2 \xRightarrow{?but \cdot !choc} \mathbf{fail} \rceil| \, k_2''$, so k_2 **fails** t_1.

We had in Example 1 that k_1, k_2 $\mathbf{io\not{c}o}$ k_3, so the erroneous k_2 is detected by t_1 but k_1 is not. The singleton test suite $\{t_1\}$ is indeed sound but not exhaustive.

4 From Model-Based Testing towards Test-Based Modelling

Practical Model-Based Testing. Model-based testing currently attracts a lot of interest, both from research and from companies. Academic as well as commercial model-based testing tools and services become available, and many companies are involved in trial projects [47,11,45,25,35,21]. This is triggered by the promise that model-based testing will make it possible to automate the testing process beyond the mere automation of test execution, which is the current state of practice in software testing. Once models are available, model-based testing allows automating the generation of test cases and the analysis of test results, thus making it possible to automate the complete testing process. Moreover, once models are available, also other sophisticated engineering and analysis methods are possible such as simulation, model checking, and implementation generation. Models, however, are not always easily available, because of various reasons.

Many modern systems are large, complex, distributed, dynamic, and networked systems, which are not monolithically built from scratch, but composed of components. Among these components are legacy, reused, general purpose, outsourced, third-party, and off-the-shelf components. These components are different in many aspects, such as different life cycles, different visibility and accessibility of internal details (black-box vs. white-box), and different forms of specifications and documentation, if documentation is available at all. For such components often no models are available, and because of insufficient documentation it is difficult or impossible to construct a valid model. Even for newly developed components, the construction of models typically requires specialized expertise and involves significant manual effort, in particular if the available documentation is poor or the knowledge about a system or component is concentrated in the minds of a few engineers. An additional issue is that systems evolve: components are substituted by newer versions or replaced by alternative components, components are restructured, or interaction with the system's environment changes. This adds maintainability of models as a main concern, and it may lead to models getting outdated as the system evolves.

The availability of models is therefore a key issue that inhibits the further proliferation of model-based testing and of other forms of model-based and model-driven analysis and development [42]. Even if models are available, they are often incomplete and not fully valid. This means that a straightforward model-based testing process consisting of sequentially performing the steps described in Sect. 2, Fig. 1, is too naive and does not work. Such a process, which would consist of sequentially making a model, generating test cases from the model, executing these test cases, and assigning a verdict, does not take into account that a discrepancy between actual outcomes and expected ones does not necessarily point to a fault in the SUT, but may be due to errors, misunderstanding, incompleteness, or invalidity of the model.

Consequently, practical model-based testing is not only checking an SUT with respect to a model, but also checking the model itself. Model-based testing in practice serves as a technique to detect discrepancies between the SUT and the model, but without making any judgment about which one is wrong. Only subsequent analysis and diagnosis can show whether the model shall be adapted, or the SUT shall be repaired. What we see is a process of concurrent improvement of both the SUT and the model by iteratively comparing them using tests: the SUT is improved using tests generated from the model, and the model is refined using observations made during these tests. The benefit of model-based testing is that this comparison is fully automated. And when in the beginning there is no model at all, this modification process starts from scratch with an 'empty' model, building up and 'learning' the model completely from observations that are made by applying tests to the SUT.

Example 6. Recently, we tested the new Dutch electronic passport [35]. Electronic passports contain a contactless smart-card that stores digitally-signed data including sensitive biometric data such as fingerprints. We developed a labelled transition system model of the electronic passport protocols, and used the model-based testing tool TorXakis to generate test cases and execute them on the actual passport. TorXakis is a straightforward implementation of the algorithms of the **ioco**-test theory of Sect. 3. It inherits from the model-based testing tool TorX [44], and adds symbolic test generation capabilities [19].

Although the behaviour of the passport is relatively simple, also in this case the most difficult part of the testing process was understanding the official specifications [24,17], which contain several hundreds of pages of detailed and wordy descriptions, and constructing a formal model from them. Access to electronic passports involves several protocols. In themselves, these protocols are fairly simple, yet, understanding their combination and interactions and extracting their essential behaviour, are a major challenge. Once we had a first model, the first test runs were more directed towards validating and checking the model and our understanding of the documents than towards testing the passport. Once there was sufficient confidence in the model the actual thorough testing of the passport took less than a week. The tests were run fully automatically; during a test run of a few days we were able to perform over 1,000,000 protocol steps on the passport. By refining and tweaking the model we could quickly learn how any underspecification and unclarities in the documents had been resolved in the implementation that we were testing.

Whereas for the passport we were eventually able to construct a valid model from the documentation, though with some difficulty, this was not the case for testing a wireless sensor network (WSN) node [50]. This experiment was carried out with the model-based test tools Uppaal-Tron [30], JTorX [5], and TorXakis. The design and development of the WSN is mainly 'guru-driven': a few clever engineers designed and developed it, and they know how it works. This implies that making a model is driven by talking with these gurus, trying to construct a model from their explanations, and subsequently trying to get their explanations confirmed by doing model-based testing experiments on the sut, which was one node of the WSN. In case of discrepancies between the model and the sut we went back to the gurus trying to get more and better explanations for these discrepancies, improved the model, and re-tested the sut. Thus, from meetings and explanations, intertwined with test experiments on the sut, we gradually and iteratively 'learned' the behaviour of the WSN node, making modifications and additions to the model in each iteration. This process is very clarifying for the testers as well as for the guru-developers who learn more about their own system.

Test-Based Modelling. On the one hand the not infrequent practice of using model-based testing to construct and refine models, and on the other hand recent theoretical developments in automata learning and grammatical inference, have led to an interesting area of research and development on the borderline of testing, verification, and machine learning, referred to as *model learning* or *test-based modelling*. Other terms are behaviour capturing [34,32], observation-based modelling [28], or just learning.

The idea of model learning is to systematically perform experiments, or tests, on a (black-box) sut, so that from the observations made during these tests a model can be constructed. It is a kind of black-box reverse engineering. Although the research area on grammatical inference and automata learning already exists for some time, only recently these techniques are supported by sufficiently powerful tools and have been applied successfully to learn models of software components.

The approach that is considered here is also called *active* or *adaptive* learning. It is adaptive because the tests used for obtaining observations are dynamically generated and optimized based on the information that has already been obtained during the learning process. It is active because through these tests extra observations are actively pursued as opposed to *passive* learning where a model is deduced solely from a set of existing system logs or traces without further interaction with the sut. The latter is possible with tools like ProM [46,28].

A number of these active learning approaches have their roots in the so-called L^* algorithm of Angluin [4]. One of these developments is an adaptation of L^* for Mealy Machines, which has been implemented in the LearnLib tool environment [39]. In the LearnLib approach a teacher, who knows a Mealy Machine model M, interacts with a learner, who wishes to learn this model. Initially only knowing the sets of input and output actions, the learner asks the teacher two kind of questions: output queries asking which output occurs in response to a particular input, and equivalence queries asking whether a particular hypothesized machine H is equivalent to the machine M. If that is the case, M has been learned and the algorithm terminates; if not the learner continues with asking more output and equivalence queries. LearnLib implements an algorithm for this learning process, i.e., a recipe for the learner which output and equivalence queries to ask, to eventually know M.

If the teacher's machine M is a real SUT, then an output query can easily be answered by the teacher by performing a test on the SUT consisting of supplying the inputs to the SUT and returning the corresponding responses. An equivalence query, however, cannot be directly answered by the teacher when M is a real black-box SUT. This is where LearnLib uses model-based testing algorithms for Mealy Machines, such as the W-method and the UIO-method, to answer whether an hypothesized machine H is equivalent to the real SUT [31].

A couple of experiments have been done with LearnLib on real SUTs [23,1], among which there is also the electronic passport of Example 6, for which a model was learned from scratch and successfully compared with the one initially developed for model-based testing [2]. These experiments show the possibilities of learning in general, and of LearnLib in particular, but as identified in [3], further work is necessary, such as: (*i*) the use of abstraction techniques to learn much larger state spaces; (*ii*) using a more general model than pure Mealy Machines which require strict alternation between inputs and outputs; and (*iii*) extension to nondeterministic systems and models.

Abstraction techniques, in particular with respect to input and output actions, have been considered in [1]. An extension to (deterministic) I/O Automata [33], which do not require the strict alternation between inputs and outputs, has been studied [3]. I/O Automata are (almost) identical to the Input-Output Transition Systems $\mathcal{IOTS}$ of Sect. 3. Nondeterminism in the context of **ioco** was elaborated in [49] where the *suspension automaton* of [41] was used as a deterministic model to represent nondeterministic labelled transition systems.

In this section we will not give new algorithms for learning or work specifically on one of these extensions. In the remainder of this section model-based testing and test-based modelling will be compared and related in the context of the abstract concepts introduced in Sect. 2, with elaborations for the **ioco** theory of Sect. 3.

Learning and Model-Based Testing. Model-based testing and learning are two sides of the same coin. Both use an SUT, a model, and tests, and aim at discovering discrepancies between behaviour described in the model and behaviour exhibited by the (black-box) SUT. Model-based testing starts with the model, and a discrepancy is in the first place considered a failure of the SUT, and an incentive to modify the SUT, after which it can be retested. Learning starts with an SUT, and a discrepancy is an incentive to adapt the hypothesized model, after which the next cycle of learning can start. But as discussed above, in practical situations the difference often disappears, as on the one hand complete and valid models for model-based testing are often lacking for various reasons, and on the other hand many learning algorithms critically depend on a model-based testing step to check an hypothesized model.

Both model-based testing and learning can be described on an abstract level by the iterative process of Fig. 6. The difference between model-based testing and learning comes from a different initial model and SUT, the use of different test generation algorithms, a different choice as what to modify in case of a discrepancy, and differences in algorithmically and manually performed steps:

– In learning, the starting SUT is given and is correct by definition. A discrepancy between model and SUT leads to adapting the model.
– In learning, the initial model can be an empty or trivial model, which can be always correct or erroneous, but it can also be an initial guess, for example, a model derived using passive testing from a set of available traces, or a model obtained in a completely different way, e.g., from reverse engineering of the code.

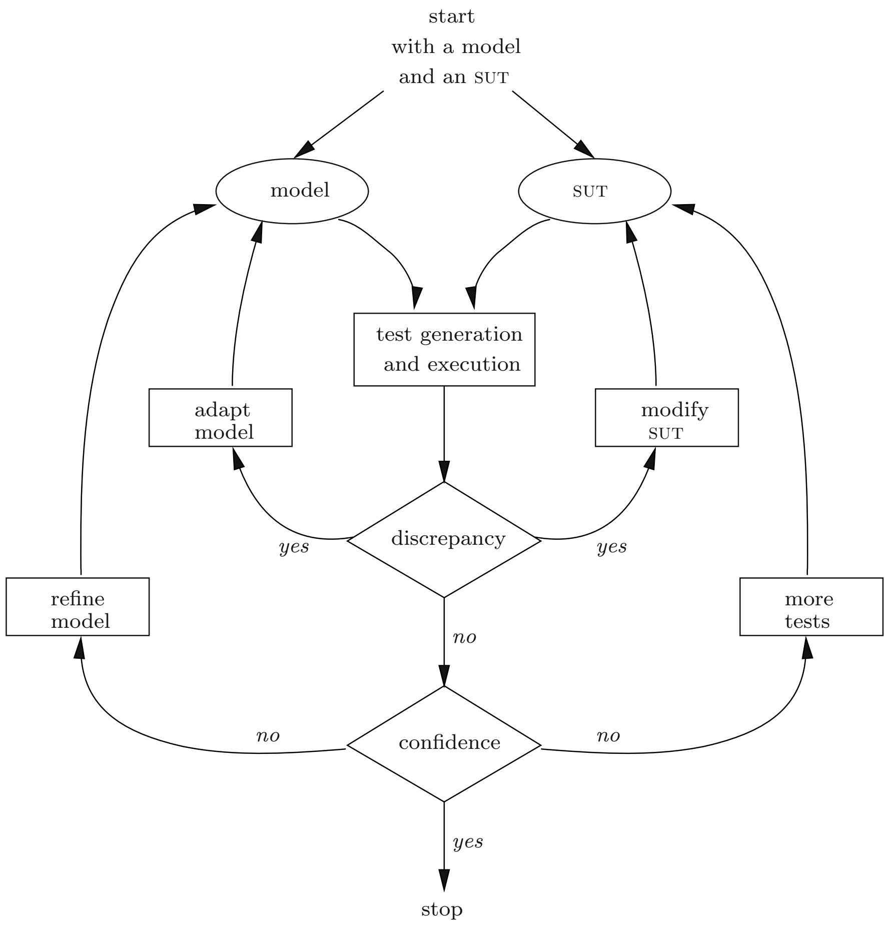


Fig. 6. The combined process of learning and model-based testing

- In learning, algorithms like Angluin-style learning take care of adapting the model completely automatically.
- A test generation algorithm for learning tries to expose as much information from the SUT as possible, i.e., it aims at rich observations with information that is useful for extending the current hypothesized model.
- If after a number of cycles in learning there is no discrepancy detected, it still can be that there is no confidence that the model is precise enough. Then the model can be refined and more tests can be executed.
- In model-based testing, the SUT is given and assumed to have been developed independently from the model.
- In (pure) model-based testing, the model is considered given and valid. Consequently, a discrepancy between model and SUT leads to modifying the SUT.
- Modifying and repairing the SUT is typically a manual step.
- A test generation algorithm for model-based testing tries to expose discrepancies between model and SUT, not necessarily providing information for extending the model.
- If after a number of cycles in model-based testing there is no discrepancy detected, it still can be that there is no confidence that the SUT is indeed correct. Then the the testing is continued with more tests.

As discussed above, current practical model-based testing processes are often a combination of learning and model-based testing. Any discrepancy has to be analysed to see whether either the model is wrong or the SUT. Such a combination completely fits in this process.

Learning in a Formal Context. We will now try to embed learning within the concepts and relations presented in Sect. 2 for model-based testing. The starting points are a formal modelling language SPEC, a domain of models of implementations MOD, an implementation relation $\mathbf{imp} \subseteq \mathit{MOD} \times \mathit{SPEC}$, and a test execution procedure leading to observations $\mathit{obs} : \mathit{TEST} \times \mathit{MOD} \to \mathcal{P}(\mathit{OBS})$.

LearnLib-style Learning in a Formal Context. In Angluin-style learning with LearnLib there is a model m which is in the class of deterministic, fully-specified Mealy Machines, which we denote with $\mathcal{MM}$. The goal of learning is also a deterministic, fully-specified Mealy Machine; thus we take $\mathit{MOD} \equiv \mathit{SPEC} \equiv \mathcal{MM}$. The tests that can applied to $\mathcal{MM}$ are sequences of input actions that lead to observations that are sequences of output actions; this defines TEST. The learning algorithm looks for a model $h \in \mathcal{MM}$ that is equivalent to m in terms of observations:

$$h \ \approx_{MM} \ m \quad \Leftrightarrow_{\text{def}} \quad \forall t \in \mathit{TEST} : \ \mathit{obs}(t,h) \ = \ \mathit{obs}(t,m) \qquad (9)$$

This means that the learning algorithm delivers the unique model m modulo $\approx_{MM}$ (which is really unique if restricted to minimal Mealy Machines). This model can be obtained in a straightforward way using all possible tests in TEST directly following (9), but fortunately LearnLib provides a more clever and efficient solution by selecting those tests that really matter based on what is already known about h.

A comparison of learning and model-based testing on $\mathcal{MM}$ shows that both are characterized by (9), the difference being the unknown variables. In testing the goal is to decide about $\approx_{MM}$ for given m and h; in learning the goal is to construct h. This leads to different algorithms, i.c. LearnLib and W- and UIO-algorithms.

Learning with Labelled Transition Systems. The next step is to consider more general models such as labelled transition systems. Such models allow arbitrary sequences of inputs and outputs instead of strict alternation, and they add nondeterminism. Arbitrary sequences of inputs and outputs were already studied in [3] in the context of I/O Automata but these systems are still deterministic.

As in Sect. 3 we consider SUTs behaving as possibly nondeterministic $\mathcal{IOTS}$, from which we wish to learn models that are labelled transition systems in $\mathcal{LTS}$, but not necessarily input-enabled: $MOD \equiv \mathcal{IOTS}(L_I, L_U)$ and $SPEC \equiv \mathcal{LTS}(L_I, L_U)$. We assume the signature of actions L_I, L_U to be known. Test cases in $TEST$ are as defined in Def. 8: $TEST \equiv \mathcal{TTS}(L_U, L_I)$, and observations and test execution are as introduced in Def. 9, i.e., $OBS \equiv L_\delta^*$ and test execution is given by $t \rceil\!| \, i \stackrel{\sigma}{\Longrightarrow} t' \rceil\!| \, i'$ where $t' =$ **pass** or **fail**.

When learning models in $\mathcal{LTS}$ from SUTs behaving as $\mathcal{IOTS}$ there are different test scenarios that can be followed depending on the relation required between the SUT and the learned model. Analogous to model-based testing various relations can be chosen, and not only equivalence of models as in the LearnLib case. Just as in testing this relation is denoted by $\mathbf{imp} \subseteq MOD \times SPEC$.

Though not the only choice, one obvious choice for **imp** in learning is the equivalence on $\mathcal{IOTS}$ induced by $TEST$, analogous as for LearnLib-style learning. Let $m \in \mathcal{IOTS}$ be the (model of) the SUT, and let $h \in \mathcal{IOTS} \subseteq \mathcal{LTS}$ be the learned model then:

$$h \;\approx_{te}\; m \quad \Leftrightarrow_{\text{def}} \quad \forall t \in \mathcal{TTS}(L_U, L_I) : \; obs(t, h) \;=\; obs(t, m) \tag{10}$$

This is the strongest relation that is testable on $\mathcal{IOTS}$ with $\mathcal{TTS}$. This implies that a learned model h satisfying (10) is the most precise model that can be obtained when learning m. The precision is expressed by the equivalence class $\{\, m' \in \mathcal{IOTS} \mid m' \approx_{te} m \,\}$. But most likely h is also the most expensive model, in terms of testing effort, that can be learned from m. And for nondeterministic transition systems there are no Learnlib-like algorithms yet that can construct h in a much more efficient way.

Another choice for **imp** is the **ioco** relation that is used for model-based testing. Then the goal is to learn a model $h \in \mathcal{LTS}$ such that m **ioco** h. There are many candidate models h that satisfy this goal. One such a model is m itself since m **ioco** m (reflexivity of **ioco** on $\mathcal{IOTS}$). Another candidate is the chaos system χ in Fig. 7, since for any m, m **ioco** χ. It even holds that m **ioco** χ_A for any m and any $A \subseteq L_I$; see Fig. 7. But χ and χ_A are not very precise and not distinctive so not very useful.

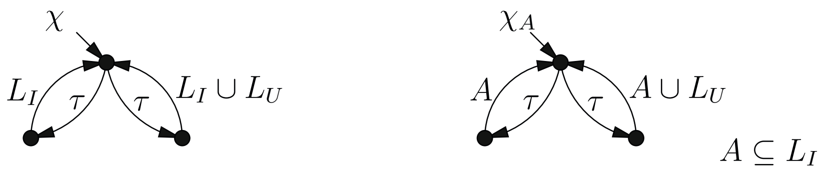


Fig. 7. Chaos. (A transition labelled with a set X is an abbreviation for all transitions with actions in that set: $q \xrightarrow{X} q' \;=_{\text{def}}\; \{q \xrightarrow{x} q' \mid x \in X\}$)

It follows that there is not a unique model that can be learned from m following the **ioco** relation but a class of models ranging from m itself to χ, and many models 'in between'. The most precise model is m itself (or more precisely its $\approx_{te}$-equivalence class), but this is also the most expensive model, in terms of test cost, to learn. The most general and therefore useless model is χ but it is also the cheapest one since it requires no testing at all. Learning can be seen as an optimization problem where we are looking for a model $h \in \mathcal{LTS}$ that optimally balances the cost of learning against obtained precision. The cost of learning a model can be expressed, for example, in the number and length of the test cases necessary to learn that model, completely analogous to Sect. 2. The precision of a learned model $h \in \mathcal{LTS}$ is related to the area of **imp**-related implementations in $\mathcal{IOTS}$. Let μ again be a monotonically increasing function on subsets of $\mathcal{IOTS}$ then the precision of a model can be expressed as $\frac{\mu(\mathcal{IOTS})-\mu(C_h)}{\mu(\mathcal{IOTS})}$ where $C_h = \{\; m' \in \mathcal{IOTS} \mid m' \textbf{ ioco } h \;\}$.

This definition of precision is analogous to the discussion on test selection for model-based testing in Sect. 2, and the measure $\mu : \mathcal{P}(MOD) \rightarrow \mathbb{R}_{\geq 0}$ is the same as the one defined there. In model-based testing, see Fig. 2, the area $P_T \backslash C_s = C_{s'} \backslash C_s$ expresses the uncertainty when using an exhaustive test suite derived from s' instead of one derived from s. Completely analogously, $C_{h'} \backslash C_h$ expresses the additional uncertainty when we stop with learned model h', i.e., we stop when we know that m **ioco** h', compared to stopping at h which may be obtained if learning is continued.

Thus, when learning a specification model h from an SUT for **ioco**-based learning there are many correct learned models. These candidate models can be compared in cost of learning and in precision. Given a measure μ on $\mathcal{IOTS}$, the precision of learned specification models can be quantified and compared. Selection of the best model is a two-dimensional optimization issue, analogous to the selection of the best test suite for model-based testing in Sect. 2.

Whereas LEARNLIB-style learning starts with an 'empty' model, which is not correct, i.e., not equivalent, **ioco** learning can start with a correct model χ which is subsequently refined and kept correct, until a sufficiently precise model is achieved, or the additional testing for a better model becomes too costly. LEARNLIB-style learning, if it succeeds, gives a very precise answer based on $\mathcal{MM}$-equivalence, but it might fail due to complexity. $\mathcal{LTS}$-style learning promises better scalability because it always gives an answer, although it might be with less precision, but it is negotiable against cost of learning.

Open Issues. This section only compared model-based testing and learning on an abstract level. No concrete algorithms for $\mathcal{LTS}$ or **ioco**-learning have been presented. Such an algorithm is, for example, presented in [49], but further elaborations are necessary.

Another one of our abstractions that obviously needs concretization is the use of the measure μ in order to quantify the over-approximation and uncertainty in testing and learning. As explained, this measure is equally applicable to expressing the quality of learned models, as well as to expressing the quality of selected test suites. Such a measure can, for example, be defined as a measure-theoretic integral over the space $\mathcal{IOTS}$ [9], but this is far from trivial. As shown in Sect. 2 such a measure can be replaced by a distance function on specification models. First attempts in this direction are [13,18].

When learning and model-based testing are combined care should be taken how to use the learned models. Doing model-based testing on a system with a fully learned model of the same system does not make sense. Testing requires independence: the SUT and the test cases, or the model from which the test cases were generated, should have been developed independently. Yet, such a learned model can be used for regression testing, i.e., testing whether a modified component still complies with the old specification. Another application domain is testing of refactored or re-implemented legacy components, which is a growing area of interest. In addition learned models can be used for other design and analysis activities, such as increasing understanding about systems, communication among stakeholders, model-based analysis of properties, model checking, and simulation.

It looks natural to extend learning of nondeterministic models with probabilities indicating the occurrence frequency of nondeterministic transitions. This opens the way towards combining with the rich area of probabilistic and stochastic state-based models.

A last point that is mentioned is uncertainty in LearnLib-style learning. This style of learning depends for its equivalence query for a real SUT on a model-based test. Model-based testing of Mealy Machines does not provide a complete decision procedure. In particular, completeness depends on the assumption that the number of states of the SUT is smaller than or equal to the number of states of the hypothesized model h. This assumption may be violated, so that testing is not complete, and consequently, the learned model is not equivalent to the SUT. Additional investigations are necessary to deal with, and quantify such incompleteness and uncertainty.

Acknowledgements. I wish to thank Fides Aarts, Brian Nielsen, Lars Frantzen, and Frits Vaandrager for discussions and for comments on earlier versions of some of the sections.

References

1. Aarts, F.: Inference and Abstraction of Communication Protocols. Master's thesis, Institute for Computing and Information Sciences, Radboud University, and Uppsala University, Nijmegen, The Netherlands, and Uppsala, Sweden (2009)

2. Aarts, F., Schmaltz, J., Vaandrager, F.: Inference and abstraction of the biometric passport. In: Margaria, T., Steffen, B. (eds.) ISoLA 2010. LNCS, vol. 6415, pp. 673–686. Springer, Heidelberg (2010)
3. Aarts, F., Vaandrager, F.: Learning I/O Automata. In: Gastin, P., Laroussinie, F. (eds.) CONCUR 2010. LNCS, vol. 6269, pp. 71–85. Springer, Heidelberg (2010)
4. Angluin, D.: Learning Regular Sets from Queries and Counterexamples. Information and Computation 75(2), 87–106 (1987)
5. Belinfante, A.: JTorX: A Tool for On-Line Model-Driven Test Derivation and Execution. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 266–270. Springer, Heidelberg (2010)
6. Berg, T., Jonsson, B., Raffelt, H.: Regular Inference for State Machines with Parameters. In: Baresi, L., Heckel, R. (eds.) FASE 2006. LNCS, vol. 3922, pp. 107–121. Springer, Heidelberg (2006)
7. Bernot, G., Gaudel, M.G., Marre, B.: Software testing based on formal specifications: a theory and a tool. Software Engineering Journal, 387–405 (November 1991)
8. Bollig, B., Katoen, J.P., Kern, C., Leucker, M.: Replaying Play in and Play out: Synthesis of Design Models from Scenarios by Learning. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 435–450. Springer, Heidelberg (2007)
9. Brinksma, E.: On the coverage of partial validations. In: Nivat, M., Rattray, C., Rus, T., Scollo, G. (eds.) AMAST 1993. BCS-FACS Workshops in Computing Series, pp. 247–254. Springer, Heidelberg (1993)
10. Brinksma, E., Tretmans, J., Verhaard, L.: A framework for test selection. In: Jonsson, B., Parrow, J., Pehrson, B. (eds.) Protocol Specification, Testing, and Verification XI, pp. 233–248. North-Holland, Amsterdam (1991)
11. Broy, M., Jonsson, B., Katoen, J.P., Leucker, M., Pretschner, A. (eds.): Model-Based Testing of Reactive Systems. LNCS, vol. 3472. Springer, Heidelberg (2005)
12. Campbell, C., Grieskamp, W., Nachmanson, L., Schulte, W., Tillmann, N., Veanes, M.: Testing Concurrent Object-Oriented Systems with Spec Explorer – Extended Abstract. In: Fitzgerald, J., Hayes, I., Tarlecki, A. (eds.) FM 2005. LNCS, vol. 3582, pp. 542–547. Springer, Heidelberg (2005)
13. Curgus, J., Vuong, S.: Sensitivity analysis of the metric based test selection. In: Kim, M., Kang, S., Hong, K. (eds.) Int. Workshop on Testing of Communicating Systems, vol. 10, pp. 200–219. Chapman & Hall, Boca Raton (1997)
14. De Nicola, R.: Extensional Equivalences for Transition Systems. Acta Informatica 24, 211–237 (1987)
15. De Nicola, R., Hennessy, M.: Testing Equivalences for Processes. Theoretical Computer Science 34, 83–133 (1984)
16. Dijkstra, E.: Notes On Structured Programming, End of section 3: On The Reliability of Mechanisms (1969)
17. Advanced Security Mechanisms for Machine Readable Travel Documents – Extended Access Control (EAC) – Version 1.11. Tech. Rep. TR-03110, German Federal Office for Information Security (BSI), Bonn, Germany (2008)
18. Feijs, L., Goga, N., Mauw, S., Tretmans, J.: Test Selection, Trace Distance and Heuristics. In: Schieferdecker, I., König, H., Wolisz, A. (eds.) Testing of Communicating Systems XIV, pp. 267–282. Kluwer Academic Publishers, Dordrecht (2002)
19. Frantzen, L., Tretmans, J., Willemse, T.: Test Generation Based on Symbolic Specifications. In: Grabowski, J., Nielsen, B. (eds.) FATES 2004. LNCS, vol. 3395, pp. 1–15. Springer, Heidelberg (2005)

20. Gaudel, M.C.: Testing can be formal, too. In: Mosses, P., Nielsen, M., Schwartzbach, M. (eds.) TAPSOFT 1995. LNCS, vol. 915, pp. 82–96. Springer, Heidelberg (1995)
21. Grieskamp, W.: Microsoft's protocol documentation program: A success story for model-based testing. In: Bottaci, L., Fraser, G. (eds.) TAIC PART 2010. LNCS, vol. 6303, pp. 7–7. Springer, Heidelberg (2010)
22. Groz, R., Charles, O., Renévot, J.: Relating Conformance Test Coverage to Formal Specifications. In: Gotzhein, R. (ed.) FORTE 1996. Chapman & Hall, Boca Raton (1996)
23. Hungar, H., Margaria, T., Steffen, B.: Domain-Specific Optimization in Automata Learning. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 315–327. Springer, Heidelberg (2003)
24. Doc 9303 – Machine Readable Travel Documents – Part 1–2. Tech. rep., ICAO, 6 edn (2006),
25. Jacky, J., Veanes, M., Campbell, C., Schulte, W.: Model-Based Software Testing and Analysis with C#. Cambridge University Press, Cambridge (2008)
26. Jard, C., Jéron, T.: TGV: Theory, Principles and Algorithms: A Tool for the Automatic Synthesis of Conformance Test Cases for Non-Deterministic Reactive Systems. Software Tools for Technology Transfer 7(4), 297–315 (2005)
27. Jeannet, B., Jéron, T., Rusu, V., Zinovieva, E.: Symbolic Test Selection based on Approximate Analysis. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 349–364. Springer, Heidelberg (2005)
28. Kanstrén, T., Piel, E., Gonzalez-Sanchez, A., Gross, H.G.: Observation-Based Modeling for Testing and Verifying Highly Dependable Systems – A Practitioner's Approach. In: Wagner, A. (ed.) Workshop on Design of Dependable Critical Systems at Safecomp 2009, Hamburg, Germany (September 2009)
29. Koopman, P., Alimarine, A., Tretmans, J., Plasmeijer, R.: Gast: Generic Automated Software Testing. In: Peña, R., Arts, T. (eds.) IFL 2002. LNCS, vol. 2670, pp. 84–100. Springer, Heidelberg (2003)
30. Larsen, K., Mikucionis, M., Nielsen, B.: Online Testing of Real-Time Systems using Uppaal. In: Grabowski, J., Nielsen, B. (eds.) FATES 2004. LNCS, vol. 3395, pp. 79–94. Springer, Heidelberg (2005)
31. Lee, D., Yannakakis, M.: Principles and Methods for Testing Finite State Machines – A Survey. The Proceedings of the IEEE 84(8), 1090–1123 (1996)
32. Lorenzoli, D., Mariani, L., Pezzè, M.: Automatic generation of software behavioral models. In: ICSE 2008: 30th Int. Conf. on Software Engineering, pp. 501–510. ACM, New York (2008)
33. Lynch, N.: Distributed Algorithms. Morgan Kaufmann Publishers Inc., San Francisco (1996)
34. Mariani, L., Pezzè, M.: Behaviour Capture and Test: Automated Analysis of Component Integration. In: 10th IEEE Int. Conf. on Engineering of Complex Computer Systems – ICECCS 2005, pp. 292–301. IEEE Computer Society, Los Alamitos (2005)
35. Mostowski, W., Poll, E., Schmaltz, J., Tretmans, J., Wichers Schreur, R.: Model-Based Testing of Electronic Passports. In: Alpuente, M., Cook, B., Joubert, C. (eds.) FMICS 2009. LNCS, vol. 5825, pp. 207–209. Springer, Heidelberg (2009)
36. Oostdijk, M., Rusu, V., Tretmans, J., de Vries, R., Willemse, T.C.: Integrating Verification, Testing, and Learning for Cryptographic Protocols. In: Davies, J., Gibbons, J. (eds.) IFM 2007. LNCS, vol. 4591, pp. 538–557. Springer, Heidelberg (2007)

37. Peled, D., Vardi, M., Yannakakis, M.: Black Box Checking. Journal of Automata, Languages, and Combinatorics 7(2), 225–246 (2002)
38. Petrenko, A.: Fault Model-Driven Test Derivation from Finite State Models: Annotated Bibliography. In: Cassez, F., Jard, C., Rozoy, B., Dermot, M. (eds.) MOVEP 2000. LNCS, vol. 2067, pp. 196–205. Springer, Heidelberg (2001)
39. Raffelt, H., Merten, M., Steffen, B., Margaria, T.: Dynamic testing via automata learning. Software Tools for Technology Transfer 11(4), 307–324 (2009)
40. Raffelt, H., Steffen, B., Berg, T., Margaria, T.: LearnLib: A framework for extrapolating behavioral models. Software Tools for Technology Transfer 11(5), 393–407 (2009)
41. Tretmans, J.: Test Generation with Inputs, Outputs and Repetitive Quiescence. Software—Concepts and Tools 17(3), 103–120 (1996)
42. Tretmans, J. (ed.): Tangram: Model-Based Integration and Testing of Complex High-Tech Systems. Embedded Systems Institute, Eindhoven (2007), `http://www.esi.nl/publications/tangramBook.pdf`
43. Tretmans, J.: Model Based Testing with Labelled Transition Systems. In: Hierons, R., Bowen, J., Harman, M. (eds.) FORTEST. LNCS, vol. 4949, pp. 1–38. Springer, Heidelberg (2008)
44. Tretmans, J., Brinksma, E.: TorX : Automated Model Based Testing. In: Hartman, A., Dussa-Zieger, K. (eds.) First European Conference on Model-Driven Software Engineering, Imbuss, Möhrendorf, Germany, December 11-12 (2003)
45. Utting, M., Legeard, B.: Practical Model-Based Testing: A Tools Approach. Morgan-Kaufmann, San Francisco (2007)
46. Verbeek, E., Buijs, J., van Dongen, B., van de Aalst, W.: Prom 6: The Process Mining Toolkit. In: 8th Int. Conf. on Business Process Management – BPM 2010 (2010)
47. de Vries, R., Belinfante, A., Feenstra, J.: Automated Testing in Practice: The Highway Tolling System. In: Schieferdecker, I., König, H., Wolisz, A. (eds.) Testing of Communicating Systems XIV, pp. 219–234. Kluwer Academic Publishers, Dordrecht (2002)
48. de Vries, R., Tretmans, J.: Towards Formal Test Purposes. In: Brinksma, E., Tretmans, J. (eds.) Formal Approaches to Testing of Software – FATES 2001. BRICS Notes Series, vol. NS-01-4, pp. 61–76. BRICS, University of Aarhus, Denmark (2001)
49. Willemse, T.: Heuristics for ioco-Based Test-Based Modelling. In: Brim, L., Haverkort, B., Leucker, M., van de Pol, J. (eds.) FMICS 2006 and PDMC 2006. LNCS, vol. 4346, pp. 132–147. Springer, Heidelberg (2007)
50. Zhu, F.: Testing Timed Systems in Simulated Time with Uppaal-Tron: An Industrial Case Study. Master's thesis, Institute for Computing and Information Sciences, Radboud University, Nijmegen, The Netherlands (2010)