Model Based Testing with Labelled Transition Systems

Jan Tretmans

Embedded Systems Institute, Eindhoven,
and Radboud University, Nijmegen,
The Netherlands
jan.tretmans@esi.nl

Abstract. Model based testing is one of the promising technologies to meet the challenges imposed on software testing. In model based testing an implementation under test is tested for compliance with a model that describes the required behaviour of the implementation. This tutorial chapter describes a model based testing theory where models are expressed as labelled transition systems, and compliance is defined with the 'ioco' implementation relation. The ioco-testing theory, on the one hand, provides a sound and well-defined foundation for labelled transition system testing, having its roots in the theoretical area of testing equivalences and refusal testing. On the other hand, it has proved to be a practical basis for several model based test generation tools and applications. Definitions, underlying assumptions, an algorithm, properties, and several examples of the ioco-testing theory are discussed, involving specifications, implementations, tests, the ioco implementation relation and some of its variants, a test generation algorithm, and the soundness and exhaustiveness of this algorithm.

1 Introduction

Software testing. Systematic testing is one of the most important and widely used techniques to check the quality of 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 consumes 30-50% of the total software development costs. The tendency is that the effort spent on testing is still increasing due to the continuing quest for better software quality, and the ever growing size and complexity of systems. The situation is aggravated by the fact that the complexity of testing tends to grow faster than the complexity of the systems being tested, in the worst case even exponentially. Whereas development and construction methods for software allow the building of ever larger and more complex systems, there is a real danger that testing methods cannot keep pace with construction. This may seriously hamper the development of future generations of software systems.

Model based testing. One of the new technologies to meet the challenges imposed on software testing is *model based testing*. In model based testing a *model* of the desired behaviour of the *implementation under test* (IUT) is the starting point for testing. Model based testing has recently gained attention with the popularization of modelling itself both in academia and in industry. The main virtue of model based testing is that it allows test automation that goes well beyond the mere automatic execution of manually crafted test cases. It allows for the algorithmic generation of large amounts of test cases, including test oracles, completely automatically from the model of required behaviour. If this model is valid, i.e., expresses precisely what the system under test should do, all these tests are also provably valid.

From an industrial perspective, model based testing is a promising technique to improve the quality and effectiveness of testing, and to reduce its cost. The current state of practice is that test automation mainly concentrates on the automatic execution of tests, but that the problem of test generation is not addressed. Model based testing aims at automatically generating high-quality test suites from models, thus complementing automatic test execution.

From an academic perspective, model based testing is a natural extension of formal methods and verification techniques, where many of the formal techniques can be reused. Formal verification and model based testing serve complementary goals. Formal verification intends to show that a system has some desired properties by proving that a model of that system satisfies these properties. Thus, any verification is only as good as the validity of the model on which it is based. Model based testing starts with a (verified) model, and then intends to show that the real, physical implementation of the system behaves in compliance with this model. Due to the inherent limitations of testing, such as the limited number of tests that can be performed, testing can never be complete: testing can only show the presence of errors, not their absence.

Sorts 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 IUT by systematically performing experiments on the real, executing IUT, 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 maintainability 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 IUT in an active way by giving stimuli and triggers to the IUT, 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 IUT should, and should not do. The specification is given in the form of some model of behaviour to which the behaviour of the IUT 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 IUT 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 IUT, 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 IUT 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.

Goal. The aim of this contribution is to be a tutorial for a particular model based testing theory, viz. the **ioco**-testing theory for labelled transition systems. This theory uses labelled transition systems as models for specifications, implementations, and tests, and a formal implementation relation called **ioco** defines conformance between implementations (IUTs) and specifications. Moreover, there is an algorithm to generate test cases, for which there is a completeness theorem (soundness and exhaustiveness) expressing that the algorithmically generated test cases exactly test for **ioco**-conformance. All of these aspects are elaborated in the following sections.

There are a couple of test generation tools, which implement, more or less directly, the **ioco**-testing theory, e.g., TVEDA [1], TGV [2], the AGEDIS TOOL SET [3], TESTGEN [4], and TORX [5]. As such, this contribution also aims at giving the theory behind these tools.

The main source for this contribution is [6]. The most important technical change with respect to [6] is the input enabledness of test cases, which was inspired by [7]; see Section 3.5.

Overview. Section 2 starts with a framework for formal, model based testing introducing the required concepts, artefacts, and relations between them. This formalism independent framework should provide a structure for discussing formal testing, and it should allow classification and comparison of different formal testing approaches. Section 3 describes the models and languages used in this contribution: labelled transition systems, and some variants of them to model specifications, implementations, and test cases. Moreover, a process language for representing them is introduced. Section 4 discusses, and formally defines the implementation relation **ioco**, which expresses what a correct implementation of a given specification should, and should not do. The algorithm for the generation of test cases from a specification is presented in Section 5. That section also considers test execution, and the correctness of the test generation algorithm. Finally, in Section 6 a few concluding remarks are given.

This contribution intends to be a tutorial accessible for anyone with some basic formal, mathematical knowledge. This implies that some readers may want to skip some of the introductory sections, in particular, the basic definitions about transition systems in Sections 3.1 and 3.2. Also Section 4.2 can be skipped; it only contains variations on the main theme which are not strictly necessary for understanding the subsequent sections.

2 Formal Testing

When performing formal, specification based testing there are different concepts and objects that we need. This section presents these concepts and objects, and the relations between them. This constitutes a kind of framework for formal testing of an implementation with respect to a formal specification of its functional behaviour. This framework, which is at a high level of abstraction, and which does not make any reference to a specific specification formalism, is depicted in Figure 1, and is explained in this section. In Sections 3, 4, and 5 these concepts will then be concretized and instantiated with the testing theory of labelled transition systems.

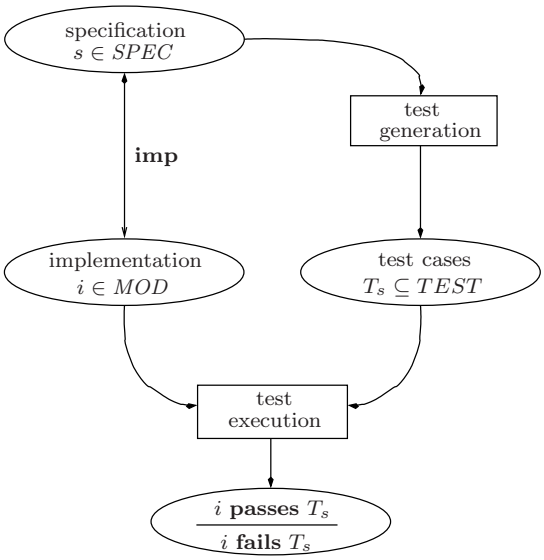


Fig. 1. The formal, specification based testing process.

Implementation. The first thing needed for testing is the *Implementation Under Test* IUT. The IUT is the system being tested. An implementation can be 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. Since we deal with black-box testing, an implementation is treated as a black-box exhibiting behaviour and interacting with its environment, but without knowledge about its internal structure. The only way a tester can control and observe an implementation is via its interfaces. The aim of testing is to check the correctness of the behaviour of the IUT on its interfaces.

Specification. Correctness of an IUT is expressed as conformance to a *specification*. The specification prescribes what the implementation should do, and what it should not do. In formal testing the specification 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 s is an element of this language: $s \in SPEC$. By means of testing we want to check whether the behaviour of the IUT conforms to s.

Conformance. To check whether the IUT conforms to a specification s we need to know precisely what it means for an IUT to conform to s, i.e., a formal definition of conformance is required. Such a definition should relate implementations to specifications. But, if we want to define such a relation between implementations and specifications, we encounter a problem. Whereas a specification s is a formal object taken from a formal domain $SPEC$, an implementation under test is not amenable to formal reasoning. An IUT is not a formal object: it is a real, physical thing, which consists of software, hardware, physical components, or a combination of these, on which only experiments and tests can be performed.

In order to formally reason about implementations we do a little trick: we make the assumption that any real implementation under test IUT can be modelled by some formal object i_{IUT} in a set of models MOD. The domain MOD is a-priori chosen, and is referred to as the universe of implementation models. This assumption is commonly referred to as the *test assumption*. Note that the test assumption presupposes a particular domain of models MOD, and that it is only assumed that a valid model i_{IUT} of the IUT exists in this domain, but not that this model i_{IUT} is a-priori known.

Thus, the test assumption allows reasoning about implementations under test as if they were formal implementations in MOD. This is what we will do from now on. Consequently, conformance can be expressed by a formal relation between models of implementations and specifications. Such a relation is called an *implementation relation* $\mathbf{imp} \subseteq MOD \times SPEC$. An implementation model i is said to be correct with respect to $s \in SPEC$ if i **imp** s.

Testing. The behaviour of an implementation is investigated by performing experiments on it. An experiment consists of supplying stimuli to the implementation and observing its responses. The specification of such an experiment, including both the stimuli and the expected responses, is called a *test case*, and it is formally expressed as an element of some domain of test cases $TEST$. The process of applying a test to an implementation is called *test execution*. Test

execution may be successful, meaning that the observed responses correspond to the expected responses, or it may be unsuccessful. The successful execution of a test t to an implementation i is expressed as i **passes** t; unsuccessful execution is denoted as i **fails** $t \Leftrightarrow i$ **pas**$\not$**ses** t. This is easily extended to a *test suite* $T \subseteq TEST$: i **passes** $T \Leftrightarrow \forall t \in T : i$ **passes** t.

Conformance testing. Conformance testing involves assessing, by means of testing, whether an implementation conforms, with respect to implementation relation **imp**, 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 from test execution an indication about conformance can be obtained. So, for conformance testing we are looking for a test suite T_s such that

$$\forall i \in MOD : \quad i \textbf{ imp } s \quad \Longleftrightarrow \quad i \textbf{ passes } T_s \tag{1}$$

A test suite with this property is said to be *complete*; it can distinguish exactly between all conforming and non-conforming implementations, i.e., testing is a complete decision procedure for **imp**-conformance to s. For practical testing this is a very strong requirement: complete test suites are infinite, and consequently not practically executable. Hence, usually a weaker requirement on test suites is posed: they should be *sound*, which means that all correct implementations, and possibly some incorrect implementations, will pass them; or, in other words, any failing implementation is indeed non-conforming, but not the other way around. Soundness corresponds to the left-to-right implication in (1). The right-to-left implication is referred to as *exhaustiveness*; it means that all non-conforming implementations are detected.

It may seem that the meaning of these concepts is reversed with respect to their usual meaning, where soundness means that no false deductions can be made, and completeness means that all correct deductions can be made. Testing, however, is about detecting errors, so that a deduction corresponds to the detection of an error. Consequently, soundness in testing means that no false deductions, i.e., no false detections of errors, can be made. Analogously, exhaustiveness (completeness) in testing means that all correct deductions can be made, i.e., that all errors can be detected.

Test generation. The systematic, algorithmic generation of test suites from a specification for a given implementation relation is called *test generation*: $gen_{\mathbf{imp}} : SPEC \rightarrow \mathcal{P}(TEST)$, (where $\mathcal{P}(TEST)$ denotes the power set of $TEST$, i.e., the set of all subsets of $TEST$). Such an algorithm is complete (sound, exhaustive) if the generated test suites are complete (sound, exhaustive) for all specifications.

Test generation is the most beneficial and visible aspect of model based testing: it allows the automatic production of large and provably sound test suites.

Conclusion. For model based testing we need a formal specification language $SPEC$, a domain of models of implementations MOD, an implementation relation **imp** $\subseteq MOD \times SPEC$ expressing correctness, a test execution procedure

passes $\subseteq MOD \times TEST$ expressing when a model of an implementation passes a test case, a test generation algorithm $gen_{\mathbf{imp}} : SPEC \rightarrow \mathcal{P}(TEST)$, and a proof that a model of an implementation passes a generated test suite if and only if it is **imp**-correct. The process of formal, specification based testing is schematically depicted in Figure 1.

The next sections will elaborate these concepts for the formalism of labelled transition systems. This means that we will use (variants of) labelled transition systems for *SPEC* (Section 3.3), *MOD* (Section 3.4), and *TEST* (Section 3.5), that conformance is expressed as a relation on labelled transition systems (Section 4), that test execution of a labelled transition system test with an implementation is defined (Section 5.1), and that a test generation algorithm is presented (Section 5.2), which is proved to generate sound and exhaustive labelled transition system test suites from a labelled transition system specification (Section 5.3).

Bibliographic notes. This framework for model based testing comes from [8, 9], with inspiration concerning test assumptions and test hypotheses from [10, 11]. There are also international standardization efforts in this direction [12]. The next sections will consider this framework instantiated with labelled transition systems, but also other formalisms may be used, e.g., Finite State Machines (FSM, Mealy Machines) [13], Abstract Data Types [11], object oriented formalisms [14], or (mathematical) functions [15].

3 Models

Model based testing uses formal specifications, models of implementations, and test case descriptions; see Figure 1. This section presents the modelling formalisms on which these specifications, models, and descriptions are built. The basic model that we use in our formal testing theory is that of a *labelled transition system*, which is defined in Section 3.1. Section 3.2 considers the representation of labelled transition systems by a formal language. Subsequently, three variants of labelled transition systems are presented: labelled transition systems with inputs and outputs (Section 3.3), input-output transition systems (Section 3.4), and test transition systems (Section 3.5), which are used to model specifications, implementations, and test cases, respectively.

3.1 Labelled Transition Systems

A labelled transition system is a structure consisting of states with transitions, labelled with actions, between them. The states model the system states; the labelled transitions model the actions that a system can perform.

Definition 1. *A* labelled transition system *is a 4-tuple* $\langle Q, L, T, q_0 \rangle$ *where*

- Q *is a countable, non-empty set of* states*;*
- L *is a countable set of* labels*;*

– $T \subseteq Q \times (L \cup \{\tau\}) \times Q$, *with* $\tau \notin L$, *is the* transition relation*;*
– $q_0 \in Q$ *is the* initial state*.*

We write $q \xrightarrow{\mu} q'$ if there is a transition labelled μ from state q to state q', i.e., $(q, \mu, q') \in T$. The informal idea of such a transition is that when the system is in state q it may perform action μ, and go to state q' . Suppose that in state q' the system can perform action μ', i.e., $q' \xrightarrow{\mu'} q''$, then these transitions can be composed: $q \xrightarrow{\mu} q' \xrightarrow{\mu'} q''$, which is written as $q \xrightarrow{\mu \cdot \mu'} q''$. In general, the composition of transitions $q_1 \xrightarrow{\mu_1 \cdot \mu_2 \cdot \ldots \cdot \mu_n} q_2$ expresses that the system, when in state q_1, can perform the sequence of actions $\mu_1 \cdot \mu_2 \cdot \ldots \cdot \mu_n$, and may end in state q_2. The use of *may* is important here: because of nondeterminism, it may be the case that the system can also perform the same sequence of actions, but end in another state: $q_1 \xrightarrow{\mu_1 \cdot \mu_2 \cdot \ldots \cdot \mu_n} q_3$ with $q_2 \neq q_3$.

Definition 2. *Let A be a set. Then A^* is the set of all finite sequences over A, with ϵ denoting the empty sequence. If $\sigma_1, \sigma_2 \in A^*$ are finite sequences, then $\sigma_1 \cdot \sigma_2$ is the concatenation of σ_1 and σ_2.*

Definition 3. *Let $p = \langle Q, L, T, q_0 \rangle$ be a labelled transition system with $q, q' \in Q$, and let $\mu, \mu_i \in L \cup \{\tau\}$.*

$$
\begin{array}{lll}
q \xrightarrow{\mu} q' & \Leftrightarrow_{\text{def}} & (q, \mu, q') \in T \\
q \xrightarrow{\mu_1 \cdot \ldots \cdot \mu_n} q' & \Leftrightarrow_{\text{def}} & \exists q_0, \ldots, q_n : \; q = q_0 \xrightarrow{\mu_1} q_1 \xrightarrow{\mu_2} \ldots \xrightarrow{\mu_n} q_n = q' \\
q \xrightarrow{\mu_1 \cdot \ldots \cdot \mu_n} & \Leftrightarrow_{\text{def}} & \exists q' : \; q \xrightarrow{\mu_1 \cdot \ldots \cdot \mu_n} q' \\
q \xrightarrow{\mu_1 \cdot \ldots \cdot \mu_n} \!\!\!\!/ & \Leftrightarrow_{\text{def}} & \textit{not } \exists q' : \; q \xrightarrow{\mu_1 \cdot \ldots \cdot \mu_n} q'
\end{array}
$$

?

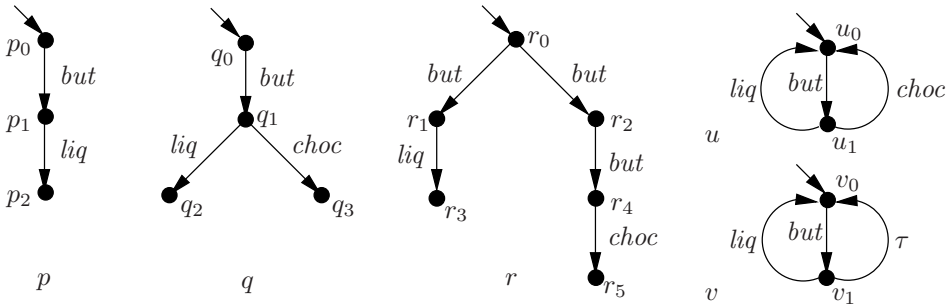


Fig. 2. Labelled transition systems.

Example 1. Figure 2 presents five examples of labelled transition systems representing candy machines. There is a button interaction *but*, and labels for chocolate *choc* and liquorice *liq*. The transition systems are represented as graphs, where nodes represent states and labelled edges represent transitions. The dangling arrow points to the initial state.

The tree q represents the labelled transition system
$\langle\, \{q_0, q_1, q_2, q_3\},\ \{but, liq, choc\},\ \{\, \langle q_0, but, q_1\rangle,\ \langle q_1, liq, q_2\rangle,\ \langle q_1, choc, q_3\rangle\, \},\ q_0\, \rangle$.
For r we have that $r_0 \xrightarrow{but} r_1$, and also $r_0 \xrightarrow{but} r_2$. Moreover, $r_0 \xrightarrow{but\cdot liq} r_3$, so also $r_0 \xrightarrow{but\cdot liq}$, but $r_0 \xrightarrow{but\cdot choc}\!\!\!\!\!\!/\ \ $.

The labels in L represent the observable actions of a system; they model the system's interactions with its environment. Internal actions are denoted by the special label τ ($\tau \notin L$), which is assumed to be unobservable for the system's environment. Also states are assumed to be unobservable for the 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 is obtained from a sequence of actions under abstraction from the internal action τ. If q can perform the sequence of actions $a{\cdot}\tau{\cdot}\tau{\cdot}b{\cdot}c{\cdot}\tau$ ($a, b, c \in L$), i.e., $q \xrightarrow{a\cdot\tau\cdot\tau\cdot b\cdot c\cdot\tau} q'$, then we write $q \xRightarrow{a\cdot b\cdot c} q'$ for the τ-abstracted sequence of observable actions. We say that q is able to perform the *trace* $a{\cdot}b{\cdot}c \in L^*$. These, and some other notations and properties are formally given in Definition 4.

Definition 4. *Let $p = \langle Q, L, T, q_0\rangle$ be a labelled transition system with $q, q' \in Q$, $a, a_i \in L$, and $\sigma \in L^*$.*

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

Example 2. In Figure 2:
$u_0 \xRightarrow{but\cdot liq\cdot but\cdot choc} u_0$, $v_0 \xRightarrow{but\cdot but\cdot but\cdot liq} v_0$, and $u_0 \overset{but\cdot but}{\Longrightarrow\!\!\!\!\!\!/}\ \ $.

In our reasoning about labelled transition systems we will not always distinguish between a transition system and its initial state. If $p = \langle Q, L, T, q_0\rangle$, we will identify the process p with its initial state q_0, and, e.g., we write $p \overset{\sigma}{\Longrightarrow}$ instead of $q_0 \xRightarrow{\sigma}$. With this in mind, we give some additional definitions and notations in Definition 5, which are exemplified in Example 3.

Definition 5. *Let p be a (state of a) labelled transition system, and $\sigma \in L^*$.*

1. $init(p) \ =_{\text{def}}\ \{\, \mu \in L \cup \{\tau\} \mid p \xrightarrow{\mu}\, \}$
2. $traces(p) \ =_{\text{def}}\ \{\, \sigma \in L^* \mid p \xRightarrow{\sigma}\, \}$
3. p **after** $\sigma \ =_{\text{def}}\ \{\, p' \mid p \xRightarrow{\sigma} p'\, \}$
4. P **after** $\sigma \ =_{\text{def}}\ \bigcup \{\, p$ **after** $\sigma \mid p \in P\, \}$, *where P is a set of states.*
5. P **refuses** $A \ =_{\text{def}}\ \exists p \in P,\ \forall \mu \in A \cup \{\tau\} :\ p \xrightarrow{\mu}\!\!\!\!\!\!/\ \ $,
   where P and A are sets of states and labels, respectively.
6. $der(p) \ =_{\text{def}}\ \{\, p' \mid \exists \sigma \in L^* :\ p \xRightarrow{\sigma} p'\, \}$
7. p *has* finite behaviour *if there is a natural number n such that all traces in $traces(p)$ have length smaller than n.*

8. p *is* finite state *if the number of reachable states* $der(p)$ *is finite.*
9. p *is* deterministic *if, for all* $\sigma \in L^*$, p **after** σ *has at most one element. If* $\sigma \in traces(p)$, *then* p **after** σ *may be overloaded to denote this element.*
10. p *is* image finite *if, for all* $\sigma \in L^*$, p **after** σ *is finite.*
11. p *is* strongly converging *if there is no state of* p *that can perform an infinite sequence of internal transitions.*
12. $\mathcal{LTS}(L)$ *is the class of all image finite and strongly converging labelled transition systems with labels in* L.

In the sequel $\mathcal{LTS}(L)$ will be our basic class of models. We restrict this class to *strongly converging* and *image finite* transition systems to make it possible to algorithmically compute an **after**-set. It will turn out that the computation of these sets is crucial for the test generation algorithm; see Section 5.2. Most of the results presented in the next sections are also valid without these restrictions, but at the expense of some extra complexity.

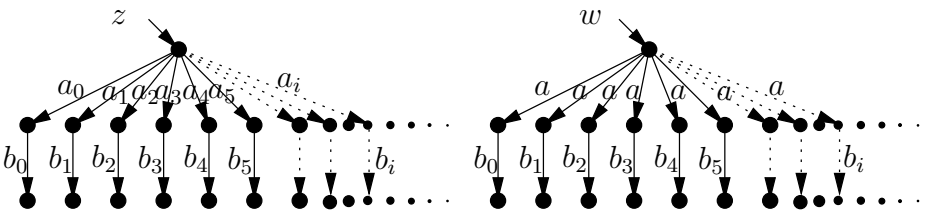


Fig. 3. An image-finite and an image-infinite transition system.

Example 3. All possible execution sequences of process r in Figure 2 are given by $traces(r) = \{\epsilon,\ but,\ but{\cdot}liq,\ but{\cdot}but,\ but{\cdot}but{\cdot}choc\}$. Process u has infinitely many traces: $traces(u) = \{\epsilon,\ but,\ but{\cdot}liq,\ but{\cdot}choc,\ but{\cdot}liq{\cdot}but,\ \ldots\}$, but although the set of traces has infinitely many elements, each trace, by definition, has finite length.

Some states in which a system can be **after** a trace are: r **after** $\epsilon = \{r_0\}$; r **after** $but = \{r_1, r_2\}$; r **after** $but{\cdot}choc = \emptyset$; v **after** $but{\cdot}liq{\cdot}but = \{v_0, v_1\}$.

We have that q **after** but **refuses** $\{but\}$, but not q **after** but **refuses** $\{liq\}$, and not q **after** but **refuses** $\{but, liq\}$. Moreover, r **after** but **refuses** $\{but\}$ and r **after** but **refuses** $\{liq\}$, but not r **after** but **refuses** $\{but, liq\}$. For v we have that $v \stackrel{but{\cdot}liq}{\Longrightarrow}$, but also v **after** but **refuses** $\{liq\}$.

The three processes p, q and r have finite behaviour and are finite state; u and v are finite state, but have infinite behaviour. Transition systems p, q, and u are deterministic, whereas r and v are nondeterministic. All five processes are image finite.

The transition system z in Figure 3 is deterministic, has infinitely many states, but has finite behaviour, and is image finite: for each $i \in \mathbb{N}$, z **after** a_i is a singleton. The system w in Figure 3 is not deterministic and not image finite: w **after** a has infinitely many states.

3.2 Representing Labelled Transition Systems

Labelled transition systems constitute a powerful semantic model to reason about processes, such as specifications, implementations, and tests. However, except for the most trivial processes, like the ones in Figure 2, an explicit representation as 4-tuple, or a representation by means of a tree or a graph is usually not feasible. Realistic systems easily have billions of states, so that drawing or enumerating them is not an option. We need another way of representing a transition system, and the usual way is to define a language with labelled transition systems as its operational semantics. Each expression in such a language defines, through its semantics, a labelled transition system. Expressions can be combined with language operators, so that complex transition systems can be composed from simpler ones. We call such a language a *process language*.

There exist many process languages. We use a variant of the language LOTOS, for which we introduce some of the constructs that are used in the sequel to define test cases, test execution, and the composition of systems. Since this text is not intended as a tutorial in such languages, we refer to the standard literature for a more detailed treatment.

The language expressions defining labelled transition systems are called *behaviour expressions*. We have the following syntax for behaviour expressions, where $a \in L$ 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*.

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

The *action prefix expression* $a\,;B$ defines the behaviour, which can perform the action a and then behaves as B, i.e., $a\,;B$ defines the labelled transition system which makes a transition labelled a to the transition system defined by B. The expression $\mathbf{i}\,;B$ is analogous to $a\,;B$, the difference being that $\mathbf{i}$ denotes the internal action τ in the transition system.

The *choice expression* $\Sigma\,\mathcal{B}$ denotes a choice of behaviour. It behaves as one of the processes in the set $\mathcal{B}$. This choice is determined by the first transition which is made. We use $B_1 \,\square\, B_2$ as an abbreviation for $\Sigma\{B_1, B_2\}$, i.e., $B_1 \,\square\, B_2$ behaves either as B_1 or as B_2. The expression **stop** is an abbreviation for $\Sigma\,\emptyset$, i.e., it is the behaviour which cannot perform any action, so it is the deadlocked process.

The *parallel expression* $B_1\,|[\,G\,]|\,B_2$ denotes the parallel execution of B_1 and B_2. In this parallel execution all actions in G must synchronize, whereas all actions not in G (including τ) can occur independently in both processes, i.e., *interleaved*. We use $\|$ as an abbreviation for $|[\,L\,]|$, i.e., sychronization on all observable actions, and $|||$ as an abbreviation for $|[\,\emptyset\,]|$, i.e., full interleaving and no synchronization.

The *hiding expression* $\mathbf{hide}\ G\ \mathbf{in}\ B$ denotes the transition system of B where all labels in G have been hidden, i.e., replaced by the internal action τ.

The last language constructs are *process definitions* and *process instantiations*. A *process definition* links a process name to a behaviour expression:

$P := B$. The name P can then be used as a process instantiation in behaviour expressions to stand for the behaviour contained in its corresponding process definition.

As usual, parentheses are used to disambiguate expressions. If no parentheses are used ';' binds stronger than '$\Box$', which binds stronger than '$|[G]|$', which in turn binds stronger than **hide**. The parallel operators are read from left to right, but note that they are not associative for different synchronization sets. So, **hide** a, c **in** $a; B_1 \;||\; b\,; B_2 \;\Box\; c; B_3 \;|||\; d; B_4$ is read as

$$(\textbf{hide}\ a, c\ \textbf{in}\ (((a; B_1)\ ||\ ((b\,; B_2)\ \Box\ (c; B_3)))\ |||\ (d; B_4)))$$

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

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

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

$$\frac{B \xrightarrow{\ a\ } B'}{\textbf{hide}\ G\ \textbf{in}\ B \xrightarrow{\ \tau\ } \textbf{hide}\ G\ \textbf{in}\ B'}\ a \in G \qquad \frac{B \xrightarrow{\ \mu\ } B'}{\textbf{hide}\ G\ \textbf{in}\ B \xrightarrow{\ \mu\ } \textbf{hide}\ G\ \textbf{in}\ B'}\ \mu \notin G$$

Table 1. Structural operational semantics.

The formal semantics of a process language is usually formally defined in the form of structural operational semantics. Such a semantic definition consists of axioms and inference rules which define for each behaviour expression the corresponding labelled transition system; see Table 1. Consider as an example the axiom for $a\,; B$. This axiom is to be read as: an expression of the form $a\,; B$ can always make a transition $\xrightarrow{\ a\ }$ to a state from where it behaves as B. Consider as another example the inference rule for $\Sigma\,\mathcal{B}$. Suppose that we can satisfy the premiss, i.e., B can make a transition labelled μ to B', and $B \in \mathcal{B}$, and μ is an observable or internal action, then we can conclude that $\Sigma\,\mathcal{B}$ can make a transition labelled μ to B'. We give the remaining axioms and rules for our language in Table 1 without further comments.

Example 4. Behaviour expressions representing the processes of Figure 2 are:

$$
\begin{array}{ll}
p & : \mathit{but};\ \mathit{liq};\ \mathbf{stop} \\
q & : \mathit{but};\ (\ \mathit{liq};\ \mathbf{stop} \ \Box\ \mathit{choc};\ \mathbf{stop}\ ) \\
r & : \mathit{but};\ \mathit{liq};\ \mathbf{stop} \ \Box\ \mathit{but};\ \mathit{but};\ \mathit{choc};\ \mathbf{stop} \\
u & : U \ \text{ where } \ U \ := \ \mathit{but};\ (\ \mathit{liq};\ U \ \Box\ \mathit{choc};\ U\ ) \\
v & : V \ \text{ where } \ V \ := \ \mathit{but};\ (\ \mathit{liq};\ V \ \Box\ \mathbf{i};\ V\ )
\end{array}
$$

These behaviour expressions are not unique, e.g., we could also choose

$$
\begin{array}{ll}
p & : \mathit{but};\ \mathit{liq};\ \mathit{liq};\ \mathbf{stop} \ ||\ \mathit{but};\ \mathit{liq};\ \mathit{choc};\ \mathbf{stop} \\
q & : \mathit{but};\ \Sigma\{\mathit{liq};\ \mathbf{stop},\ \mathit{choc};\ \mathbf{stop}\}
\end{array}
$$

The parallel operator in particular can be used to efficiently represent large transition systems. Consider the process p of Figure 2 which has 3 states. The interleaving of p with itself, $p \,|||\, p$, has $3 \times 3 = 9$ states; and $p \,|||\, p \,|||\, p$ has 27 states, and so forth.

Also infinite-state processes can be represented with finite expressions, e.g., Y with $Y \ := \ a \ ;\ (\ b \ ;\ \mathbf{stop} \ |||\ Y\ )$ has infinitely many states; it can perform actions a and b but it can never do more b's than a's. The infinite-state transition system z in Figure 3 can be written as $\Sigma\ \{\ a_i \ ;\ b_i \ ;\ \mathbf{stop}\ \mid\ i \in \mathbb{N}\ \}$.

Finally, note that not every behaviour expression represents a transition system in $\mathcal{LTS}(L)$, e.g., the image-infinite transition system w of Figure 3 can be expressed in our language: $\Sigma\{\ a \,;\, b_i \,;\, \mathbf{stop}\ \mid\ i \in \mathbb{N}\ \}$. Also the transition system defined by $\mathbf{hide}\ a\ \mathbf{in}\ \ P_0$, where $P_i \ := \ a \,;\, P_{i+1} \ \Box\ b_i \,;\, \mathbf{stop}$ with $i \in \mathbb{N}$, is not in $\mathcal{LTS}(L)$; it is neither image finite, nor strongly converging. In the rest of this paper we will not bother about the semantics being only partially defined.

3.3 Inputs and Outputs

A labelled transition system defines the possible sequences of interactions that a system may have with its environment. These interactions are abstract, in the sense that they are only identified by a label; there is no notion of initiative or direction of the interaction, nor of input or output. An interaction can occur if both the process and its environment are able to perform that interaction, implying that they can also both block the occurrence of the interaction. The communication between a system and its environment is symmetric. When the environment is also a labelled transition system this communication can be expressed in our language by the parallel synchronization operator $|[\,G\,]|$, where the labels in G model the possible interactions.

Although this paradigm of abstract interaction is sufficient for analysing and reasoning about many applications, there are also systems which communicate in a different way, in particular those systems that we will consider for testing. Those systems do not abstract from initiative and direction; they do distinguish between actions initiated by the environment, and actions initiated by themselves. They communicate via *inputs* and *outputs*: outputs are actions initiated by the system, and inputs are actions initiated by the environment.

We define *labelled transition system with inputs and outputs* to model systems for which the set of actions is partitioned into input actions contained in an input label set L_I, and output actions in an output label set L_U. (The 'U' refers to 'uitvoer', the Dutch word for 'output', which is preferred to avoid confusion between L_O (letter 'O') and L_0 (digit zero)).

Definition 6. *A* labelled transition system with inputs and outputs *is a 5-tuple* $\langle Q, L_I, L_U, T, q_0 \rangle$ *where*

- $\langle Q, L_I \cup L_U, T, q_0 \rangle$ *is a labelled transition system in* $\mathcal{LTS}(L_I \cup L_U)$*;*
- L_I *and* L_U *are countable sets of* input labels *and* output labels*, respectively, which are disjoint:* $L_I \cap L_U = \emptyset$.

The class of labelled transition systems with inputs in L_I *and outputs in* L_U *is denoted by* $\mathcal{LTS}(L_I, L_U)$.

Inputs are usually decorated with '?' and outputs with '!'. Labelled transition systems with inputs and outputs are used as formal specifications in our testing theory, i.e., $\mathcal{LTS}(L_I, L_U)$ instantiates the class of specifications *SPEC*; see Section 2. Of course, this does not mean that specifications have to be written explicitly as labelled transition systems: any language with labelled transition system semantics suffices, for example, the process language defined in Section 3.2.

3.4 Input-Output Transition Systems

Labelled transition systems with inputs and outputs do not really differ from normal labelled transition systems. We define *input-output transition systems* to model systems with inputs and outputs, in which outputs are initiated by the system and never refused by the environment, and inputs are initiated by the system's environment and never refused by the system. This means that the system is always prepared to perform any input action, i.e., all inputs are enabled in all states.

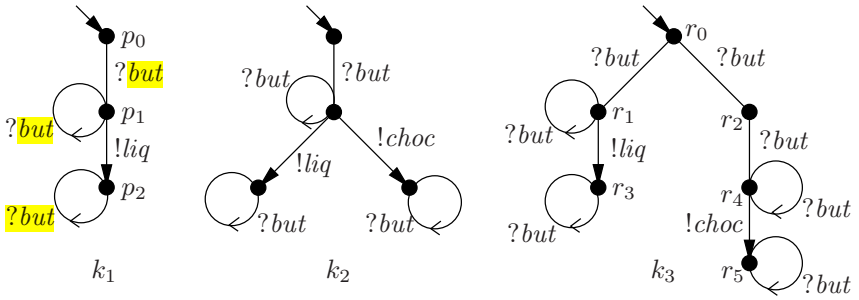


Fig. 4. Input-output transition systems.

Definition 7. *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:*

$$\forall q \in der(q_0), \forall a \in L_I : \; q \stackrel{a}{\Longrightarrow}$$

The class of input-output transition systems with inputs in L_I *and outputs in* L_U *is denoted by* $\mathcal{IOTS}(L_I, L_U) \subseteq \mathcal{LTS}(L_I, L_U)$.

Example 5. In Figure 2 only v is an input-output transition system, when $L_I = \{?but\}$ and $L_U \supseteq \{!liq\}$. Some other input-output transition systems are given in Figure 4. In k_1 we can push the button $?but$, which is an input for the candy machine, and then the machine outputs liquorice $!liq$. After $?but$ has been pushed once, and also after the machine has released $!liq$, any more pushing of $?but$ has no effect: k_1 makes a self-loop and does not change state. In fact, k_1, k_2, and k_3 are almost the same transition systems as p, q, and r in Figure 2, interpreting $L_I = \{?but\}$ as inputs and $L_U = \{!liq, !choc\}$ as outputs, and the difference being that self-loop transition have been added to make them input enabled; this adding of input self-loops is sometimes referred to as *angelic completion*.

Another way of making systems input-enabled is to add a special error state, and to add transitions to this error state for all non-specified inputs. In Figure 5 yet another completion method, viz. *demonic completion*, is used to make u in Figure 2 input enabled: all non-specified inputs lead to the *chaos* process χ. Once in χ any behaviour is possible.

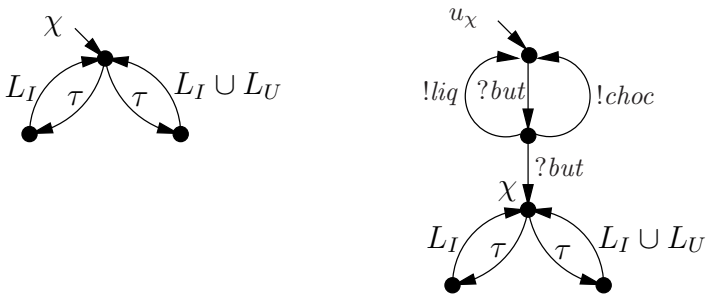


Fig. 5. Demonic completion.

Since input-output transition systems are just a special kind of labelled transition systems, all definitions for them apply. In particular, the parallel synchronization operator $\|$ is used to model the communication between a system s and its environment e. Naturally, the set L_I of inputs of s should correspond to the outputs of e, and the set L_U of outputs of s are the inputs of e: $e \in \mathcal{IOTS}(L_U, L_I)$. Since the inputs of s are always enabled, e can autonomously

determine whether an action in L_I will occur, or not. Conversely, since all actions in L_U are always enabled in e, it is up to s to determine whether an action in L_U occurs, or not. If s cannot perform an output action, it can only wait until e performs one of e's outputs. Such a state without output actions, where s cannot autonomously proceed, is called *suspended*, or *quiescent*. A quiescent state q is denoted as $\delta(q)$. If, and only if, both s and e are quiescent then there is no way to proceed: they are in *deadlock*.

A system s with environment e is in deadlock after trace σ if there is no possible action to proceed: $(s \parallel e)$ **after** σ **refuses** L . For non-input-enabled transition systems this means

$$\exists A_s, A_e \subseteq L : A_s \cup A_e = L \quad \text{and} \quad s \textbf{ after } \sigma \textbf{ refuses } A_s \quad \text{and} \quad e \textbf{ after } \sigma \textbf{ refuses } A_e \tag{2}$$

For input-output transition systems this is simplified, since s can never refuse actions in L_I: if s **after** σ **refuses** A_s then $A_s \subseteq L_U$. Analogously, if e refuses something it must be a subset of L_I. Since $L_I \cap L_U = \emptyset$, the only sets satisfying (2) are $A_s = L_U$ and $A_e = L_I$, i.e., when both s and e are quiescent: $\delta(s)$ and $\delta(e)$.

Although this rationale for introducing quiescence is based on input-output transition systems, the definition applies equally well to non-input-output transition systems, and since in subsequent sections we will indeed consider quiescence also for non-input-enabled transition systems, it is generally defined in Definitions 8 and 9 for labelled transitions with inputs and outputs.

Definition 8. *Let* $p \in \mathcal{LTS}(L_I, L_U)$.

1. *A state* q *of* p *is* quiescent, *denoted by* $\delta(q)$, *if* $\forall \mu \in L_U \cup \{\tau\} : q \not\xrightarrow{\mu}$
2. *The* quiescent traces *of* p *are those traces that may lead to a quiescent state:* $Qtraces(p) =_{\text{def}} \{ \sigma \in L^* \mid \exists p' \in (p \textbf{ after } \sigma) : \delta(p') \}$

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. It turns out to be convenient to express this 'seeing nothing', i.e., quiescence, as a special 'output action'; it is denoted by δ ($\delta \notin L \cup \{\tau\}$). Once we have this special action we can also consider transitions with δ. Such a transition $p \xrightarrow{\delta}$ expresses that p allows the observation of quiescence, i.e., p cannot perform any output action. In this way the absence of outputs is made into an explicitly observable event. Since quiescence implies that no real transition is performed, the goal state after a δ-transition is always the same as the start state, so $p \xrightarrow{\delta} p$ if $\delta(p)$.

With δ-transitions it is also possible to extend traces with δ. For example, $p \xRightarrow{\delta \cdot ?a \cdot \delta \cdot ?b \cdot !x}$ 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. Traces that may contain the quiescence action δ, are called *suspension traces*.

Definition 9. *Let* $p = \langle Q, L_I, L_U, T, q_0 \rangle \in \mathcal{LTS}(L_I, L_U)$.

1. $L_\delta \ =_{\text{def}} \ L \cup \{\delta\}$
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* $Straces(p) \ =_{\text{def}} \ \{ \ \sigma \in L_\delta^* \mid p_\delta \stackrel{\sigma}{\Longrightarrow} \ \}$

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 3, 4, and 5 also apply to transition systems with label set L_δ.

In our testing theory it will be assumed that an implementation under test can be modelled as an input-output transition system, i.e., $\mathcal{IOTS}(L_I, L_U)$ instantiates the class of models of implementations MOD, where L_I and L_U are assumed to be the same as given for the specification; see Section 2. Since models of implementations are only assumed to exist, the language representation issue does not play a role for implementations. Whereas implementations are input-enabled, specifications are not necessarily input-enabled. This difference allows having partial specifications; this will be elaborated in Section 4, in particular in Examples 9 and 10.

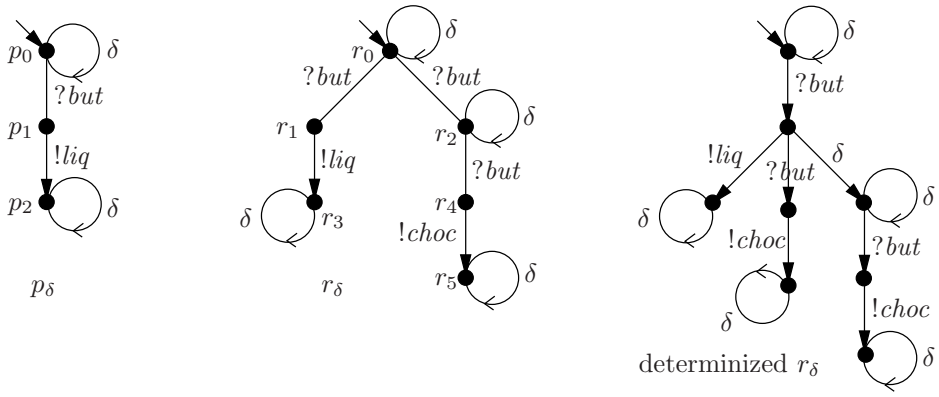


Fig. 6. Quiescence.

Example 6. For k_1 in Figure 4 we have that $\delta(p_0)$, $\delta(p_2)$, but not $\delta(p_1)$ because state p_1 can perform output $!liq$.

As explained above, the definition of quiescence is not restricted to input-enabled systems. It can be applied to any transition system with inputs and outputs. If in Figure 2 we take $L_I = \{?but\}$ and $L_U = \{!liq, !choc\}$, then for process p we have: $\delta(p_0)$, $\delta(p_2)$, but not $\delta(p_1)$.

In Figure 6, quiescence has been made explicit by adding the δ-transitions for p and r of Figure 2. So, we have, for example, $p_\delta \stackrel{\delta \cdot ?but \cdot !liq \cdot \delta \cdot \delta}{\Longrightarrow}$, and the suspension trace $?but \cdot \delta \cdot ?but \cdot !choc \in Straces(r)$, but $?but \cdot \delta \cdot ?but \cdot !liq \notin Straces(r)$.

Since $\delta(r_2)$ but not $\delta(r_1)$ we know that we are in the right branch after having observed $?but{\cdot}\delta$. In the determinization of r_δ this is even more explicit: after the sequence $?but{\cdot}\delta$ the continuation is $?but{\cdot}!choc$. This determinization of r_δ, which may serve as a kind of canonical representation of transition systems modulo equality of suspension traces, is sometimes referred to as the *suspension automaton* of r. In such a determinization a δ-transition is not necessarily a loop anymore, and an output- and δ-transition may be enabled in one state. Determinizing a transition system to which δ-transitions have been added is not the same as adding δ-transitions to a determinized transition system. Moreover, there are deterministic transition systems over $L_I \cup L_U \cup \{\delta\}$ for which there is no labelled transition system for which it is the suspension automaton.

3.5 Test Cases

A test case is a specification of the behaviour of a tester in an experiment carried out on an implementation under test. In this experiment the tester serves as a kind of artificial environment of the implementation. Following the discussion in Section 3.4 about the communication between an implementation and its environment, an implementation can do three different things: it can accept any input in L_I, it can produce an output in L_U, or it can remain quiescent. This implies that the tester, being this environment, must provide these inputs, must be able to observe these outputs, and must be able to observe quiescence if there is no output. Moreover, the tester should be input-enabled for all actions in L_U. The behaviour of such a tester is also modelled as an input-output transition system, but, naturally, with inputs and outputs exchanged. For observing quiescence, we add a special label θ to the transition systems modelling tests ($\theta \notin L_I \cup L_U \cup \{\tau, \delta\}$). The occurrence of θ in a test indicates the detection of quiescence δ, i.e., the observation that no output is produced by the implementation. Theoretically, this means that the tester has to wait for an infinite amount of time in order to conclude that implementation does not, and will never produce any output. More practically, one could think of θ as being implemented as the expiration of a time-out. Of course, care should be taken when choosing such a (finite) time-out value in order to have confidence that after an output-less time-out period the system indeed is quiescent.

Combining the above, we have that test cases are in the first place processes in $\mathcal{IOTS}(L_U, L_I \cup \{\theta\})$. But, based on the observation that the execution of a test case is an experiment under control of the tester, a few restrictions are added. First, there must be a mechanism in test cases to assign verdicts. This is accomplished by having two special verdict states called **pass** and **fail**, which are sink states, i.e., once in **pass** (**fail**) the test case cannot leave that state anymore. Second, in order to make it possible to assign a verdict within finite time, test cases should always allow reaching a **pass** or **fail** state within finitely many transitions. Third, in order to keep the tester in control, unnecessary nondeterminism should be avoided. In the first place, this implies that the test case itself is deterministic. In the second place, this means that a tester should never offer more than one input action (from the perspective of the implementation) at a

time. Since the implementation is able to accept any input action, offering more inputs would always lead to an unnecessarily nondeterministic continuation of the test run. Having a deterministic test case does not imply that a test run has a unique result: due to nondeterminism in the implementation under test, and due to nondeterminism in the test run itself, the repetition of a test run may lead to a different result; see also Section 5.1. Altogether, we come to the following definition of the class of test transition systems $\mathcal{TTS}$, which instantiates the domain of test cases $TEST$; see Section 2.

Since the use of 'input' and 'output' in a test case is always confusing (and it will be even more confusing once we start the discussion on test execution, where implementations and tests come together), we try to use the convention that 'input' and 'output' always refer to the inputs and outputs of the specification and implementation under test. Consequently, input-enabledness of a test case means that all actions in L_U are enabled. Also the decorations '?' and '!' refer to the use of actions in the specification (or implementation). The special action θ is indeed special: it is considered neither input, nor output.

Definition 10.

1. *A* test case *t for an implementation with inputs in L_I and outputs in L_U is an input-output transition system $\langle Q, L_U, L_I \cup \{\theta\}, T, q_0 \rangle \in \mathcal{IOTS}(L_U, L_I \cup \{\theta\})$ such that*
   - *t is finite state and deterministic;*
   - *Q contains two special states* **pass** *and* **fail**, **pass** $\neq$ **fail**, *with*
     $\mathbf{pass} := \Sigma\, \{\ x\,;\, \mathbf{pass} \mid x \in L_U \cup \{\theta\}\ \}$
     $\mathbf{fail} := \Sigma\, \{\ x\,;\, \mathbf{fail} \mid x \in L_U \cup \{\theta\}\ \}$
   - *t has no cycles except those in states* **pass** *and* **fail**
     (formally: for $\sigma \in (L \cup \{\theta\})^* \setminus \{\epsilon\}$: $q \stackrel{\sigma}{\Longrightarrow} q$ implies *$q =$* **pass** or **fail***)*
   - *for any state $q \in Q$ of the test case*
     either $init(q) = \{a\} \cup L_U$ for some $a \in L_I$
     or $init(q) = L_U \cup \{\theta\}$.
2. *The class of test cases for implementations with inputs L_I and outputs L_U is denoted as $\mathcal{TTS}(L_U, L_I)$.*
3. *A* test suite *T is a set of test cases: $T \subseteq \mathcal{TTS}(L_U, L_I)$.*

Example 7. Figure 7 gives two example test cases with $L_I = \{?but\}$ and $L_U = \{!liq, !choc\}$. Test case t_1 provides input $?but$ to an implementation. If this is successful t_1 expects to receive $!liq$ from the implementation followed by nothing, i.e., quiescence. Any other reaction is considered erroneous and leads to **fail**.

In test case t_2 the loops in the states **pass** and **fail** have been omitted. Since they are only there to make the test case input enabled, we will omit them from now on.

3.6 Some Bibliographic Notes

Labelled transition systems are a basic model in formal theories. Many languages for processes, in particular process algebras, use transition systems for their

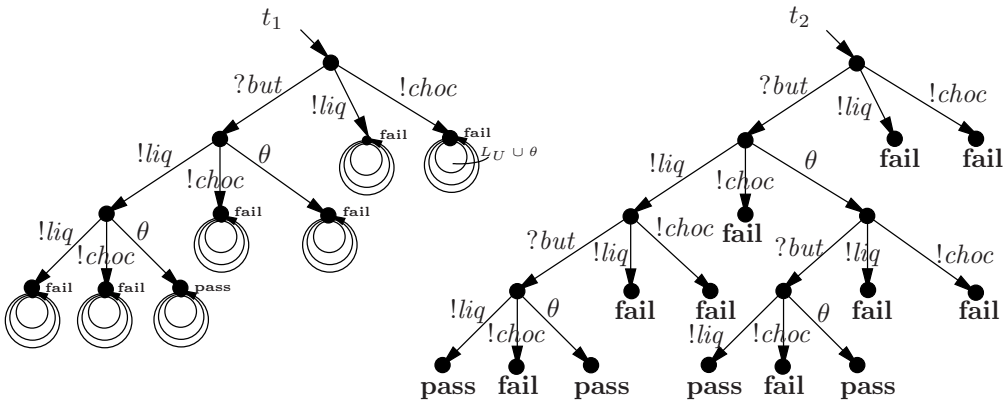


Fig. 7. Test cases.

operational semantics, which is usually defined through structured operational semantics; [16] may serve as a literature entry. The language presented here is mainly inspired by LOTOS [17, 18].

Input-output transition systems are analogous to Input/Output Automata (IOA) [19], and to Input-Output State Machines [1], but they differ from classical Finite State Machines or Mealy Machines. Input-output transition systems differ marginally from Input/Output Automata in input enabling: instead of requiring *strong input enabling* as in [19] ($\forall a \in L_I : p' \xrightarrow{a}$), input-output transition systems allow input enabling via internal transitions (*weak input enabling*, $\forall a \in L_I : p' \stackrel{a}{\Longrightarrow}$). Moreover, as is usual for labelled transition systems, all internal actions are denoted by the same label τ, which is possible, since we do not consider fairness explicitly, so we do not need to express a partitioning of output and internal actions to pose fairness requirements as in Input/Output Automata. Testing is only about finite behaviours.

4 The Implementation Relation

The purpose of this section is to define precisely when an implementation in $\mathcal{IOTS}(L_I, L_U)$ is correct with respect to a specification in $\mathcal{LTS}(L_I, L_U)$, i.e., to define an implementation relation; see Section 2. There are many ways of defining an implementation relation. In principle, any relation between $\mathcal{IOTS}(L_I, L_U)$ and $\mathcal{LTS}(L_I, L_U)$ could serve as an implementation relation, but, of course, some relations appear more natural and intuitive than others. The main relation that we consider is called **ioco**, and it is defined in Section 4.1. Subsequently, Section 4.2 discusses some variants and properties of **ioco**.

4.1 The Implementation Relation *ioco*

The implementation relation on which we build our test theory is **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 from i that is foreseen by s. A special output of i is the absence of outputs as modelled by quiescence δ; see Section 3.4. This means that if i is quiescent then s should have the possibility to be quiescent, too. A formal definition of **ioco** starts with defining the set *out* of possible outputs. This set can contain the special label δ. It is defined for a single state, and then generalized to a set of states. The latter is used in combination with **after** (Definition 5.3): $out(\, p \,\mathbf{after}\, \sigma \,)$ gives all possible outputs occurring after having performed the trace $\sigma \in L_\delta^*$.

Definition 11. *Let q be a state in a transition system, and let Q be a set of states, then*

1. $out(q) \;=_{\text{def}}\; \{\; x \in L_U \mid q \xrightarrow{x} \;\} \;\cup\; \{\; \delta \mid \delta(q) \;\}$
2. $out(Q) \;=_{\text{def}}\; \bigcup \{\; out(q) \mid q \in Q \;\}$

Example 8. Some examples for k_3 in Figure 4:

$$
\begin{array}{lll}
out(\, k_3 \,\mathbf{after}\, \epsilon \,) & = out(r_0) & = \{\delta\} \\
out(\, k_3 \,\mathbf{after}\, \delta \,) & = out(r_0) & = \{\delta\} \\
out(\, k_3 \,\mathbf{after}\, !liq \,) & = out(\emptyset) & = \emptyset \\
out(\, k_3 \,\mathbf{after}\, ?but \,) & = out(r_1) \cup out(r_2) & = \{!liq, \delta\} \\
out(\, k_3 \,\mathbf{after}\, ?but{\cdot}?but \,) & = out(r_1) \cup out(r_4) & = \{!liq, !choc\} \\
out(\, k_3 \,\mathbf{after}\, ?but{\cdot}\delta{\cdot}?but \,) & = out(r_4) & = \{!choc\} \\
out(\, k_3 \,\mathbf{after}\, ?but{\cdot}?but{\cdot}!liq \,) & = out(r_3) & = \{\delta\} \\
out(\, k_3 \,\mathbf{after}\, ?but{\cdot}\delta{\cdot}?but{\cdot}!liq \,) & = out(\emptyset) & = \emptyset
\end{array}
$$

The informal idea that 'any output produced by i has been foreseen in s' is formally expressed by requiring that the *out*-set of the implementation is a subset of the *out*-set of the specification. This should hold for any state, but due to nondeterminism we do not exactly know in which state we are during testing. What can be observed is the suspension trace $\sigma \in L_\delta^*$ executed so far, which may lead to different states. The set $p \,\mathbf{after}\, \sigma$ collects all these possible current states, so $out(\, p \,\mathbf{after}\, \sigma \,)$ contains all possible outputs, possibly including δ, after σ. This set, when obtained from the implementation, must be included in the analogous set obtained from the specification, but only if σ is a suspension trace of the specification. Altogether, these considerations lead to Definition 12.

Definition 12. *Given a set of input labels L_I and a set of output labels L_U, the relation* **ioco** $\subseteq \mathcal{IOTS}(L_I, L_U) \times \mathcal{LTS}(L_I, L_U)$ *is defined as follows:*

$$i \;\mathbf{ioco}\; s \quad \Leftrightarrow_{\text{def}} \quad \forall \sigma \in Straces(s) : \quad out(\, i \,\mathbf{after}\, \sigma \,) \;\subseteq\; out(\, s \,\mathbf{after}\, \sigma \,)$$

The fact that **ioco** only requires inclusion of *out*-sets for the suspension traces of the specification, together with the fact that specifications can be non-input enabled, makes it possible to have *partial* specifications. For suspension traces which are not in $Straces(s)$ there is no requirement whatsoever on the implementation, which implies that an implementation is free to implement anything it likes after such a trace. Such a trace is said to be *underspecified*. In many situations it can be beneficial to a have a partial specification, whereas in other situations a *complete* specification is preferred: a complete specification specifies after every trace what should happen. Completeness can always be achieved with **ioco** by having an input enabled specification: if $s \in \mathcal{IOTS}(L_I, L_U)$ then there are no underspecified traces.

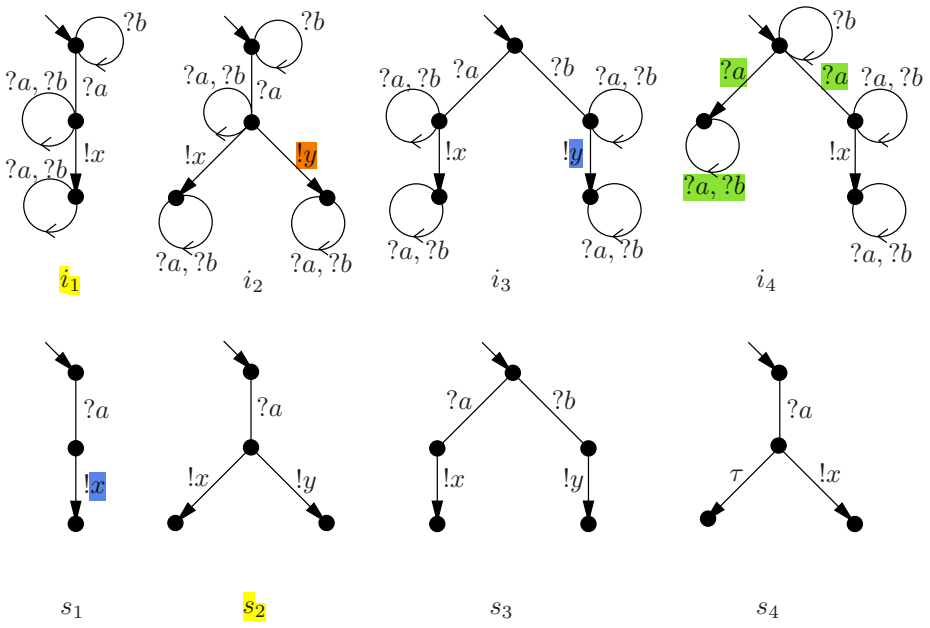


Fig. 8. Implementations and specifications with **ioco**.

Example 9. Figure 8 gives some implementations and specifications with $L_I = \{?a, ?b\}$ and $L_U = \{!x, !y\}$:

i_m **ioco** s_n	s_1	s_2	s_3	s_4
i_1	**ioco**	**ioco**	**io⊬co**	**ioco**
i_2	**io⊬co**	**ioco**	**io⊬co**	**io⊬co**
i_3	**ioco**	**ioco**	**ioco**	**ioco**
i_4	**io⊬co**	**io⊬co**	**io⊬co**	**ioco**

Specification s_1 specifies that after input $?a$ output $!x$ must occur, which is expressed as: $out(\, s_1 \textbf{ after } ?a \,) = \{x\}$. Implementation i_1 satisfies this require-

ment, but i_2 and i_4 do not: $out(\, i_2 \textbf{ after } ?a\,) = \{x, y\} \not\subseteq \{x\}$, $out(\, i_4 \textbf{ after } ?a\,) = \{x, \delta\} \not\subseteq \{x\}$. For i_3, $out(\, i_3 \textbf{ after } ?a\,) = \{x\} \subseteq out(\, s_1 \textbf{ after } ?a\,)$. Moreover, $out(\, i_3 \textbf{ after } ?b\,) = \{y\} \not\subseteq out(\, s_1 \textbf{ after } ?b\,) = \emptyset$, but since $?b \notin Straces(s_1)$ this does not matter, and hence i_3 **ioco** s_1.

We see from i_2 **io$\not$co** s_1 that an implementation should not produce more outputs than allowed by the specification, and from i_4 **io$\not$co** s_1 that the implementation should not be quiescent, when the specification expects an output. But from i_3 **ioco** s_1 we see that an implementation may have additional features, in this case the behaviour $?b{\cdot}!y$; the specification is *partial*, or *underspecified* for $?b$: s_1 does not prescribe any requirements for behaviour that follows $?b$, and an implementation is completely free to do anything it likes after $?b$.

Specification s_2 requires that after input $?a$ either output $!x$ or $!y$ is performed. This means that i_1, i_2, i_3 **ioco** s_2, but not i_4 **ioco** s_2, since i_4 may not produce any output at all: $out(\, i_4 \textbf{ after } ?a\,) = \{x, \delta\} \not\subseteq \{x, y\}$.

We see from i_1 **ioco** s_2 that an implementation may have less outputs than the specification allows, but having no output at all is not allowed.

Specification s_3 specifies that output $!x$ must be performed after $?a$, and $!y$ after $?b$. Implementations i_1, i_2, and i_4 are quiescent after $?b$, so they are not **ioco**-correct; i_3 does satisfy this requirement.

Specification s_4 specifies that after $?a$ either output $!x$ should be produced, or the implementation may be quiescent: $out(\, s_4 \textbf{ after } ?a\,) = \{x, \delta\}$. Only i_2 may produce output $!y$ and is not **ioco**-correct.

The implementations can also be mutually compared. Of course, i_k **ioco** i_k for $k = 1, 2, 3, 4$, since **ioco** is reflexive on $\mathcal{IOTS}(L_I, L_U)$. Furthermore, only i_1 **ioco** i_2, i_4. Mutually comparing the specifications does not make sense, since the specifications are not input enabled, so **ioco** is not defined.

Example 10. Consider Figure 4. We have that k_1 **ioco** k_2: an implementation capable of only producing $!liq$ conforms to a specification that prescribes to produce either $!liq$ or $!choc$. Although k_2 is deterministic according to Definition 5.9, in fact, it specifies in an input-output context that after $?but$ there is a nondeterministic choice between supplying $!liq$ or $!choc$.

If we want to specify a machine that produces both *liquorice* and *chocolate*, then two buttons are needed to select the respective candies, cf. s_3 in Example 9:

$$?liq\text{-}button \;;\; !liq \;;\; \textbf{stop} \;\;\square\;\; ?choc\text{-}button \;;\; !choc \;;\; \textbf{stop}$$

On the other hand, k_2 **io$\not$co** k_1 and k_2 **io$\not$co** k_3: if the specification prescribes to produce only $!liq$ then an implementation shall not have the possibility to produce $!choc$.

We have k_1 **ioco** k_3, but k_3 **io$\not$co** k_1 and k_3 **io$\not$co** k_2, since k_3 may refuse to produce anything after the *button* has been pushed once, whereas both k_1 and k_2 will always output something; formally: $\delta \in out(\, k_3 \textbf{ after } ?but\,)$, whereas $\delta \notin out(\, k_1 \textbf{ after } ?but\,)$ and $\delta \notin out(\, k_2 \textbf{ after } ?but\,)$.

Figure 2 contains three non-input-enabled transition systems, which may serve as specifications. We have k_1 **ioco** p, and k_2 **io$\not$co** p. Also p is underspecified:

p does not specify what should happen after the *button* has been pushed twice, since $?but{\cdot}?but \notin Straces(p)$.

Moreover, k_1 **ioco** q and k_2 **ioco** q, but k_3 **io$\not$o** p and k_3 **io$\not$o** q. As before, this is the case because $\delta \in out(\, k_3 \textbf{ after } ?but\,)$, whereas $\delta \notin out(\, p \textbf{ after } ?but\,)$ and $\delta \notin out(\, q \textbf{ after } ?but\,)$.

4.2 Some Variations and Properties of *ioco*

Generalization. The implementation relation **ioco** (Definition 12) requires that the *out*-set of the implementation be a subset of the specification's *out*-set for all traces in the set of suspension traces of the specification. By making this set of suspension traces a parameter of the relation a family of implementation relations is defined.

Definition 13. *Let $\mathcal{F} \subseteq (L_I \cup L_U \cup \{\delta\})^*$ be a set of suspension traces, $i \in \mathcal{IOTS}(L_I, L_U)$, and $s \in LTS(L_I, L_U)$.*

$$i \textbf{ ioco}_{\mathcal{F}} s \quad \Leftrightarrow_{\text{def}} \quad \forall \sigma \in \mathcal{F} : \quad out(\, i \textbf{ after } \sigma\,) \subseteq out(\, s \textbf{ after } \sigma\,)$$

Typically, the set $\mathcal{F} \subseteq L_\delta^*$ depends on the specification s. Clearly, i **ioco** s iff i **ioco**$_{Straces(s)}$ s, but also some other relations for specific $\mathcal{F}$ have been given names, and, based on sub-setting of the respective sets $\mathcal{F}$ these relations can be easily compared.

Definition 14. *Let $i \in \mathcal{IOTS}(L_I, L_U)$, $s \in LTS(L_I, L_U)$.*

1. $i \leq_{ior} s \;=_{\text{def}}\; i \textbf{ ioco}_{L_\delta^*} s$
   iff $\forall \sigma \in L_\delta^* : \; out(\, i \textbf{ after } \sigma\,) \subseteq out(\, s \textbf{ after } \sigma\,)$
2. i **ioconf** $s \;=_{\text{def}}\; i \textbf{ ioco}_{traces(s)} s$
   iff $\forall \sigma \in traces(s) : \; out(\, i \textbf{ after } \sigma\,) \subseteq out(\, s \textbf{ after } \sigma\,)$
3. $i \leq_{iot} s \;=_{\text{def}}\; i \textbf{ ioco}_{L^*} s$
   iff $\forall \sigma \in L^* : \; out(\, i \textbf{ after } \sigma\,) \subseteq out(\, s \textbf{ after } \sigma\,)$

Proposition 1.

1. *$\leq_{ior}$ and $\leq_{iot}$ are preorders on $\mathcal{IOTS}(L_I, L_U)$, i.e., they are reflexive and transitive when restricted to input-enabled transition systems.*
2. $\mathcal{F}_1 \subseteq \mathcal{F}_2$ implies $\textbf{ioco}_{\mathcal{F}_1} \supseteq \textbf{ioco}_{\mathcal{F}_2}$
3. $\leq_{ior} \subset \left\{ \begin{matrix} \leq_{iot} \\ \textbf{ioco} \end{matrix} \right\} \subset \textbf{ioconf}$
4. $i \leq_{ior} s$ iff $Straces(i) \subseteq Straces(s)$.
5. $i \leq_{iot} s$ iff $traces(i) \subseteq traces(s)$ and $Qtraces(i) \subseteq Qtraces(s)$

Example 11. The difference between $\leq_{iot}$ and $\leq_{ior}$, and between **ioconf** and **ioco** is illustrated with the processes r_1 and r_2 in Figure 9: r_1 **ioconf** r_2, but r_1 **io$\not$o** r_2; in terms of *out*-sets: $out(\, r_1 \textbf{ after } ?but{\cdot}\delta{\cdot}?but\,) = \{!liq, !choc\}$ and $out(\, r_2 \textbf{ after } ?but{\cdot}\delta{\cdot}?but\,) = \{!choc\}$.

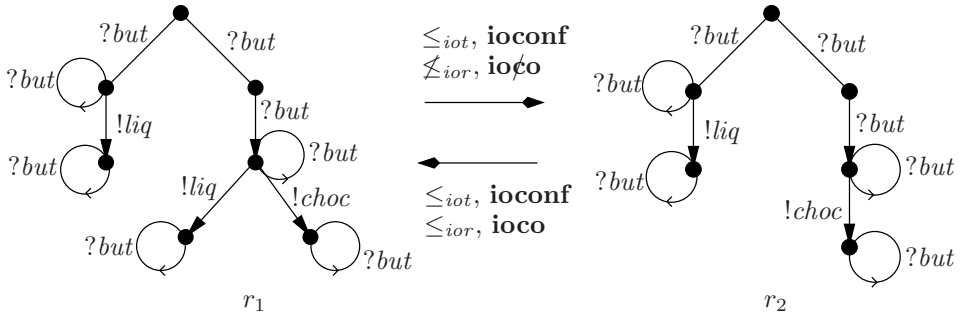


Fig. 9. The difference between $\leq_{iot}$ and $\leq_{ior}$

Intuitively, after pushing the *button*, we observe that nothing is produced by the machine, so we push the *button* again. Machine r_1 may then produce either *liquorice* or *chocolate*, while machine r_2 will always produce *chocolate*. When we use the relation **ioconf**, the observation always terminates after observing that nothing is produced; quiescence can only be an element of the *out*-set, but it cannot occur in the trace leading to the state where the *out*-set is calculated. Hence, there is no way to distinguish between entering the left or the right branch of r_1 or r_2; after the *button* is pushed twice, both machines may produce either *liquorice* or *chocolate*: $out(\, r_{1,2}\, \textbf{after}\, ?but{\cdot}?but\,) = \{!liq, !choc\}$.

Partial specifications. In Section 4.1 it was mentioned that two conditions make it possible to have *partial* specifications, viz. first, that **ioco** only requires inclusion of *out*-sets for the suspension traces of the specification, and, second, that specifications are non-input enabled.

If the first condition is changed to inclusion of *out*-sets for all possible suspension traces in L^*_δ, the relation $\leq_{ior}$ is obtained; see Definition 14.1. From Proposition 1.4 it follows that $\leq_{ior}$ indeed does not allow partiality: all behaviours of a $\leq_{ior}$-correct implementation, as expressed by its suspension traces, are contained in those of the specification. This is also valid for underspecified specifications in $\mathcal{LTS}(L_I, L_U) \backslash \mathcal{IOTS}(L_I, L_U)$, which implies that it does not make sense to have an underspecified specification in combination with $\leq_{ior}$.

With respect to the second condition, if specifications are input enabled, i.e., **ioco** is restricted to a relation *on* $\mathcal{IOTS}(L_I, L_U)$, and there are no underspecified traces anymore, then what remains turns out to be exactly the relation $\leq_{ior}$.

Proposition 2.

1. $\sigma \in Straces(p)$ iff $out(\, p\, \textbf{after}\, \sigma\,) \neq \emptyset$
2. *If* $i, s \in \mathcal{IOTS}(L_I, L_U)$ *then* i **ioco** s iff $i \leq_{ior} s$
3. *If* $p, q \in \mathcal{IOTS}(L_I, L_U)$, *and* $s \in \mathcal{LTS}(L_I, L_U)$ *then* p **ioco** q and q **ioco** s imply p **ioco** s.
4. **ioco** *is a preorder on* $\mathcal{IOTS}(L_I, L_U)$.

Underspecified traces and uioco. Another relation in the family $\mathbf{ioco}_{\mathcal{F}}$ is **uioco**. For the rationale for **uioco** consider r in Figure 2 as a specification with $L_I = \{?but\}$ and $L_U = \{!liq, !choc\}$. Since r is not input enabled, it is a partial specification. For example, $?but{\cdot}?but{\cdot}?but$ is an underspecified trace, and any 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 $out(\, r \,\mathbf{after}\, ?but{\cdot}?but\,) = \{!choc\}$. But suppose that in the first $?but$-transition r moves nondeterministically to state r_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., see Definition 15.1, 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.

An alternative characterization of **uioco** can be given by transforming a partial specification into an input enabled one with demonic completion using the chaos process χ, as explained in Example 5. In this way the specification makes explicit that after an underspecified trace anything is allowed.

Definition 15. *Let $i \in \mathcal{IOTS}(L_I, L_U)$, and $s \in \mathcal{LTS}(L_I, L_U)$.*

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

Example 12. Because $Utraces(s) \subseteq Straces(s)$ it is clear (proposition 1.2) that **uioco** is not stronger than **ioco**. That it is strictly weaker follows from the following example. Take r in Figure 2 as (partial) specification, and consider r_1 and r_2 from Figure 9 as potential implementations. Then $r_2\;\mathbf{ioco}\;r$ because $!liq \in out(\, r_2\,\mathbf{after}\,?but{\cdot}?but\,)$ and $!liq \notin out(\, r\,\mathbf{after}\,?but{\cdot}?but\,)$, but $r_2\;\mathbf{uioco}\;r$ because $?but{\cdot}?but \notin Utraces(r)$. Also $r_1\;\mathbf{ioco}\;r$, but in this case also $r_1\;\mathbf{uioco}\;r$ because $?but{\cdot}\delta{\cdot}?but \in Utraces(r)$, $!liq \in out(\, r_1\,\mathbf{after}\,?but{\cdot}\delta{\cdot}?but\,)$ and $!liq \notin out(\, r\,\mathbf{after}\,?but{\cdot}\delta{\cdot}?but\,)$.

Variants. A couple of other variations on **ioco** have been defined:

TGV-ioco TGV is a tool for the automatic synthesis of test cases for nondeterministic systems [2]. Its underlying theory is analogous to the **ioco** theory with a small extension. TGV deals with divergencies or livelocks consisting of τ-loops. Such a livelock is given an *unfair* semantics, which means that the system can loop for ever. This implies that an external observer will not see any progress in the system, i.e., the observer will see quiescence.

mioco The relation multi-**ioco** extends **ioco** with multiple channels [20]. Each action belongs to exactly one input channel or output channel. Each output channel can be quiescent, and moreover each input channel can be blocked meaning that the channel (temporarily) does not accept any inputs. Analogous to **ioco**, **mioco** requires that the outputs, output quiescences, and input blockings occurring in an implementation, are included in those of the specification.

(r)tioco Different(real)-timed-**ioco** relations have been defined: [21–23]. The difficulty, and the difference between the different versions of timed-**ioco** is the treatment of quiescence. Quiescence in **ioco** means that no outputs are produced, not now and not in the future. If time is an explicit parameter in the models, it is also possible to require and observe that no outputs are produced for a specified period of time, whereas 'in the future' should be defined more precisely with explicit mentioning of time.

ioco$_r$ Sometimes, the level of granularity of the actions in the implementation is different from that of the specification, i.e., the label sets L_I and L_U of specification and implementation cannot be assumed to be the same. Usually, this means that one abstract action of the specification is implemented by a sequence of actions in the implementation. This is called action refinement, and leads to a relation **ioco**$_r$ [24].

sioco Actions are sometimes parameterized with data. In order to avoid state explosion during test generation, this data is treated in a symbolic way, leading to a symbolic-**ioco** [25]. Whereas the other variants above exend or alter **ioco**, **sioco** does not change it; it only gives another representation of the relation in case data variables and parameters are involved.

hioco Hybrid systems are systems in which discrete actions and continuous variables play a role. Ongoing research aims at defining a relation hybrid-**ioco** to formalize the relation between a hybrid transition system implementation and its specification.

Compositional testing. With the popularization of component based development it is desirable that also testing and integration can be based on components, i.e., that successfully tested components can be integrated into correctly functioning systems. Unfortunately, this is not directly the case for **ioco**-correctness: the composition of two **ioco**-correct implementations i_1 and i_2, communicating through actions in V, and modelled as **hide** V **in** $i_1 \; |[V]| \; i_2$, is not necessarily **ioco**-correct to the composition of their specifications; see Proposition 3.1. Technically, this means that **ioco** is not a precongruence for the hiding and parallel operators. Intuitively, it can be understood by seeing that a component's specification may have underspecified traces after which the component's implementation may show any possible behaviour. In a composition with another component this behaviour may lead to undesired behaviour which is not allowed by the composition of the specifications. This problem can be avoided by having no underspecified traces: the precongruence property does hold for input enabled specifications; see Proposition 3.2.

Proposition 3. *Let* $i_k \in \mathcal{IOTS}(L_{Ik}, L_{Uk})$ *and* $s_k \in \mathcal{LTS}(L_{Ik}, L_{Uk})$ *for* $k = 1, 2$*, be two componenents, such that they have disjoint inputs and disjoint outputs:* $L_{I1} \cap L_{I2} = L_{U1} \cap L_{U2} = \emptyset$*. Let* $V = (L_{I1} \cap L_{U2}) \cup (L_{U1} \cap L_{I2})$ *be the set of their common interactions.*

1. i_1 **ioco** s_1 *and* i_2 **ioco** s_2 *does* not *imply*
   (**hide** V **in** $i_1 \,|[\, V \,]|\, i_2$) **ioco** (**hide** V **in** $s_1 \,|[\, V \,]|\, s_2$)
2. *If* s_1, s_2 *are input-enabled, i.e.,* $s_k \in \mathcal{IOTS}(L_{Ik}, L_{Uk})$ *for* $k = 1, 2$*; then*
   i_1 **ioco** s_1 and i_2 **ioco** s_2 *implies that*
   (**hide** V **in** $i_1 \,|[\, V \,]|\, i_2$) **ioco** (**hide** V **in** $s_1 \,|[\, V \,]|\, s_2$)

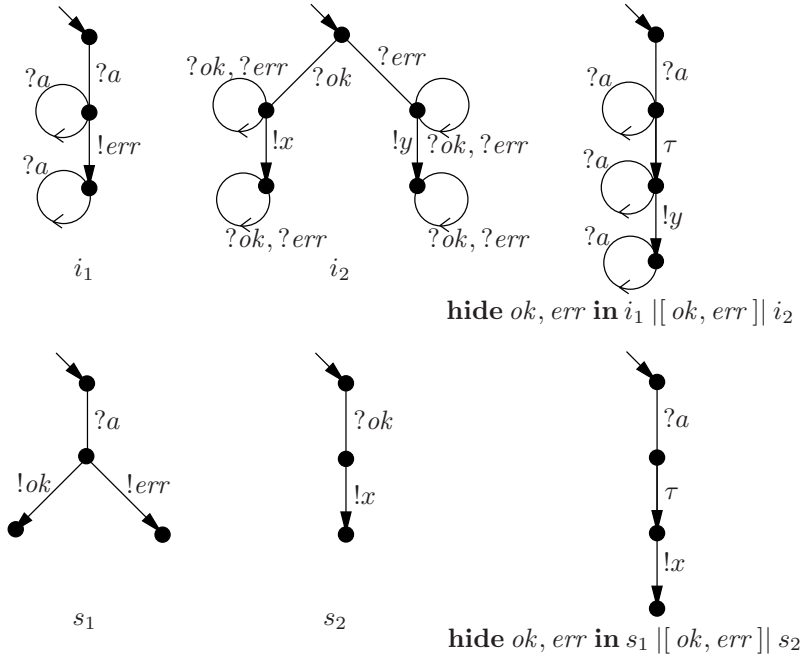


Fig. 10. Compositional testing.

Example 13. Consider specifications s_1 and s_2 and implementations i_1 and i_2 in Figure 10. The input set for s_1 and i_1 is $L_{I1} = \{a\}$; the outputs of s_1 and i_1 are equal to the inputs of s_2 and i_2: $L_{U1} = L_{I2} = \{ok, err\}$; the outputs of s_2 and i_2 are $L_{U2} = \{x, y\}$.

Specification s_1 specifies that upon input $?a$ an output $!ok$ or $!err$ shall be produced, which is **ioco**-correctly implemented in i_1. The partial specification s_2 specifies that on input $?ok$ the output $!x$ shall be provided, which i_2 correctly implements; hence, i_1 **ioco** s_1 and i_2 **ioco** s_2. But upon composing i_1 and i_2 the additional behaviour of i_2, viz. providing output $!y$ for input $?err$ that is underspeci-

fied in s_2, causes incorrect behaviour with respect to the composition of the specifications: (**hide** ok, err **in** $i_1 |[ok, err]| i_2$) **ioco** (**hide** ok, err **in** $s_1 |[ok, err]| s_2$).

4.3 Some Bibliographic Notes

The relation **ioco** inherits many ideas from other relations on transition systems defined in the literature. Its roots are in the theory of testing equivalence and preorders [26, 27], where the relation testing preorder on transitions systems is defined by explicitly introducing the environment, or tester, and the observations that a tester can make of a system. Some developments, which build on these testing preorders, are of importance for **ioco**. In the first place, there were the introduction of more powerful testers, which can detect not only the occurrence of actions but also the absence of actions, i.e., refusals, in [28], and the addition of a special label θ to observe refusals in [29]. A second development was a testing theory based on testing preorders, where a conformance relation **conf**, and test generation algorithms were defined by restricting all observations of testing preorder testers to those traces that are explicitly contained in the specification [30]. A third development was the application of the principles of testing preorder to Input/Output Automata (IOA) in [31], where it was shown that testing preorder coincides with quiescent trace preorder [32] when requiring that inputs are always enabled. The relation **ioco** inherits from all these developments. The definition of **ioco** follows the principles of testing preorder with tests that can also detect the refusal of actions. Outputs and always enabled inputs are distinguished analogous to IOA, and, moreover, a restriction is made to only the traces of the specification as in **conf**.

Whereas this section first presented **ioco** and then introduced $\mathbf{ioco}_{\mathcal{F}}$, $\leq_{ior}$, $\leq_{iot}$, and **ioconf** as variants, the historical development, and the way they were presented in [6], were the other way around: it started with $\leq_{iot}$, and then **ioconf**, $\leq_{ior}$, **ioco**, and $\mathbf{ioco}_{\mathcal{F}}$ followed. Moreover, the original definitions are given as testing relations as in [26, 27], and the definitions in this text were propositions in [6].

Another interesting historical event was the development of the testing theory for the tool TVEDA [1], which occurred independently and without reference to an underlying theory of testing equivalence or refusal testing, but only based on intuition and formalization of existing protocol testing practice and experience. This resulted in a relation called R_1 which strongly resembles **ioco**. This may be another indication, apart from all case studies done since then, of the practical relevance of **ioco**.

The relation **uioco**, also called $\mathbf{ioco}_U$, was the result of studying congruence and compositionality properties for **ioco** in [33]. The anomaly of nondeterministic underspecified traces was also remarked in [7, 34].

Most of the complete proofs for **ioco** can be found in [6, technical report version].

5 Testing with Labelled Transition Systems

Now that we have formal specifications, implementations, test cases, and the implementation relation **ioco** expressing correctness, we can start the discussion on testing. We are looking for a test generation algorithm that derives a set of tests from a specification, so that by executing these tests we know, or at least can get an indication, whether an implementation **ioco**-conforms to that specification. For that, we first have to discuss what test execution is, and what it means to pass a test. This is done in Section 5.1. Then the test generation algorithm is given in Section 5.2. Subsequently, Section 5.3 shows that this algorithm has the required correctness properties, that is, the generated test suites detect all and only non-conforming implementations. This means that such a generated test suite can serve as a decision procedure for **ioco**-conformance.

5.1 Test Execution

A test run of a test case $t \in \mathcal{TTS}(L_U, L_I)$ with an implementation under test $i \in \mathcal{IOTS}(L_I, L_U)$ is an experiment where the test case supplies inputs to the implementation, while observing the outputs, and the absence of outputs (quiescence) of the implementation. This might be described as the parallel synchronization $t \parallel i$ (Section 3.2), but this does not take into account the peculiarities of the special labels δ and θ. Hence, we extend $\parallel$ to $\rceil\!\!\mid\!\!\mid$ to take into account that θ is used to observe quiescence δ; see Definition 16.1.

A test run $t \rceil\!\!\mid\!\!\mid i$ can always continue, i.e., it has no deadlocks. This follows from the construction of a test case; see Definition 10.1: for each state t' of a test case either $init(t') = \{a\} \cup \{L_U\}$ for some $a \in L_I$, or $init(t') = L_U \cup \{\theta\}$. In the former case the action a can always be performed on the implementation since i is input enabled. In the latter case either i produces some output $x \in L_U$, or i is quiescent. In both cases the test run can continue, be it with an infinite sequence of θ actions. Since **pass** and **fail** are sink states a test run can be stopped if one of these is reached. The trace of $t \rceil\!\!\mid\!\!\mid i$ to that point identifies the test run; it can be seen as the test log of the test run.

Since an implementation can behave nondeterministically, different test runs of the same test case with the same implementation may lead to different terminal states, and hence to different verdicts. An implementation passes a test case if and only if all possible test runs lead to the verdict **pass**. This means that each test case must be executed several times in order to explore all possible nondeterministic behaviours of the implementation, and, moreover, that a particular fairness must be assumed on implementations, i.e., it is assumed that an implementation by re-execution of a test case shows all its possible nondeterministic behaviours with that test case.

Definition 16. *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 $\rceil\!\!\mid\!\!\mid : \mathcal{TTS}(L_U, L_I) \times \mathcal{IOTS}(L_I, L_U) \rightarrow \mathcal{LTS}(L_I \cup L_U \cup \{\theta\})$ which*

is defined by the following inference rules:

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

2. *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:*

   σ *is a test run of t and i* $\Leftrightarrow_{\text{def}}$ $\exists i' :\ t\rceil\!|\, i \stackrel{\sigma}{\Longrightarrow} \mathbf{pass}\rceil\!|\, i'$ or $t\rceil\!|\, i \stackrel{\sigma}{\Longrightarrow} \mathbf{fail}\rceil\!|\, i'$

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

$$i\ \mathbf{passes}\ t \quad \Leftrightarrow_{\text{def}} \quad \forall \sigma \in L_\theta^*,\ \forall i' :\ t\rceil\!|\, i \stackrel{\sigma}{\not\Longrightarrow} \mathbf{fail}\rceil\!|\, i'$$

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 the test suite, it fails: $i\ \mathbf{fails}\ T \ \Leftrightarrow_{\text{def}}\ \exists t \in T : i\ \mathbf{passes}\!\!\not\ t$.

Example 14. Consider the test cases in Figure 7 and the implementations in Figure 4. The only test run of t_1 with k_1 is $t_1\rceil\!|\, k_1 \xrightarrow{?but\cdot!liq\cdot\theta} \mathbf{pass}\rceil\!|\, k_1'$, so k_1 **passes** t_1.
For t_1 with k_2 there are two test runs:
$t_1\rceil\!|\, k_2 \xrightarrow{?but\cdot!liq\cdot\theta} \mathbf{pass}\rceil\!|\, k_2'$, and $t_1\rceil\!|\, k_2 \xrightarrow{?but\cdot!choc} \mathbf{fail}\rceil\!|\, k_2''$, so k_2 **fails** t_1.
Also k_3 **fails** t_1: $t_1\rceil\!|\, k_3 \xrightarrow{?but\cdot!liq\cdot\theta} \mathbf{pass}\rceil\!|\, k_3'$, but also $t_1\rceil\!|\, k_3 \xrightarrow{?but\cdot\theta} \mathbf{fail}\rceil\!|\, k_3''$.
When t_2 is applied to k_3 we get:
$t_2\rceil\!|\, k_3 \xrightarrow{?but\cdot!liq\cdot?but\cdot\theta} \mathbf{pass}\rceil\!|\, k_3'$, $t_2\rceil\!|\, k_3 \xrightarrow{?but\cdot\theta\cdot?but\cdot!choc} \mathbf{fail}\rceil\!|\, k_3''$, so k_3 **fails** t_2.

5.2 Test Generation

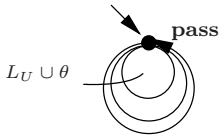
Now all ingredients are there to present an algorithm to generate test cases from a labelled transition system specification, which test implementations for **ioco**-correctness. To see how such test cases may be constructed, we consider the definition of **ioco**; see Definition 12. We see that to test for **ioco** we have to check whether $out(\, i\ \mathbf{after}\ \sigma\,) \subseteq out(\, s\ \mathbf{after}\ \sigma\,)$ for each $\sigma \in traces(s)$. Basically, this can be done by having a test case t that executes σ: $t\rceil\!|\, i \stackrel{\sigma}{\Longrightarrow} t'\rceil\!|\, i'$. After this the test case should check whether the produced outputs by i' are allowed by s. This can be done by having transitions from t' to **pass**-states for all allowed outputs – those in $out(\, s\ \mathbf{after}\ \sigma\,)$ – and transitions to **fail**-states for all erroneous outputs – those not in $out(\, s\ \mathbf{after}\ \sigma\,)$. Special care should be taken for the special output δ: δ models the absence of any output, which matches with the θ-transition in the test case. Consequently, the θ-transition will go the **pass**-state if quiescence is allowed – $\delta \in out(\, s\ \mathbf{after}\ \sigma\,)$ – and to the **fail**-state if the specification does not allow quiescence at that point.

All this is reflected in the following test generation algorithm. The algorithm is recursive: the first transition of the test case is derived from the states in which the specification can initially be, after which the remaining part of test case is recursively derived from the specification states reachable from the inital states via this first test case transition. The algorithm is nondeterministic in the sense that in each recursive step it can be continued in many different ways: the test case can be terminated with the test case **pass** (choice 1); the test case can continue with any input allowed by the specification, which can be interrupted by an arriving output (choice 2); or the test case can wait for an output and check it, or conclude that the implementation is quiescent (choice 3). Each choice for continuation results in another, valid test case. Also here the set L_U, i.e., the specification's outputs, contains the inputs of the generated test case, and L_I its outputs.

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

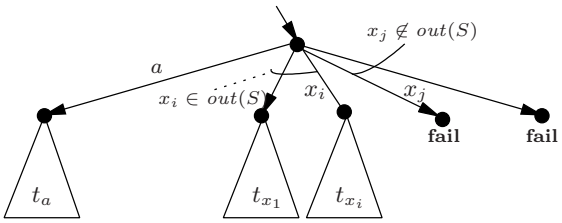
A test case $t \in \mathcal{TTS}(L_U, L_I)$ is obtained from a non-empty set of states S by a finite number of recursive applications of one of the following three nondeterministic choices:

1.



$t :=$ **pass**

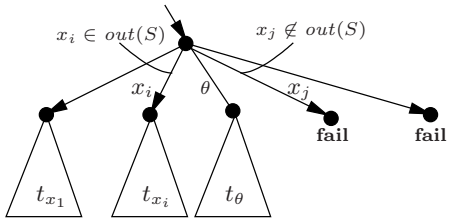
2.



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

*where $a \in L_I$ such that S **after** $a \neq \emptyset$, t_a is obtained by recursively applying the algorithm for the set of states S **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 **after** x_i .*

3.



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

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

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, this is the set s **after** $\epsilon = q_0$ **after** ϵ, where q_0 is the initial state of s. Then the test case is built step by step. In each step there are three ways to make a test case:

1. The first choice is the single-state test case **pass**, which is always a sound test case. It stops the recursion in the algorithm, and thus terminates the test case.
2. In the second choice test case t attempts to supply input a to the implementation, and subsequently behaves as test case t_a. Test case t_a is obtained by recursive application of the algorithm with the set S **after** a, which is the set of specification states that can be reached via an a-transition from some current state in S. Moreover, t is prepared to accept any output of the implementation (not quiescence) that might occur before a is supplied. Analogous to t_a, each t_{x_i} is obtained from S **after** x_i.
3. The third choice consists of checking the next output of the implementation. In this case the test case does not attempt to supply an input; it waits until an output arrives, and if no output arrives it observes quiescence. If the response, whether a real output or quiescence, is not allowed, i.e., $x_j \notin out(S)$, 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 **after** x_i.

Example 15. Test case t_1 of Figure 7 can be obtained from specification p in Figure 2 using Algorithm 1:

1. Initially, $S = p$ **after** $\epsilon = \{p_0\}$.
2. Choice 2 is made, i.e., we try to give an input to the implementation. The only input with S **after** $a \neq \emptyset$ is $?but$, so $t_1 := ?but; t_1^2 \Box\ !liq; \mathbf{fail} \Box\ !choc; \mathbf{fail}$.
3. To obtain t_1^2, the next output of the implementation is checked (choice 3): $t_1^2 := !liq; t_1^3 \Box\ !choc; \mathbf{fail} \Box\ \theta; \mathbf{fail}$.

4. For t_1^3 the output is checked again (choice 3), where now the only allowed response is quiescence: $t_1^3 := !liq; \mathbf{fail} \,\square\, !choc; \mathbf{fail} \,\square\, \theta; t_1^4$.
5. For t_1^4 we stop (choice 1): $t_1^4 := \mathbf{pass}$.

After putting all pieces together, we obtain t_1 of Figure 7 as a test case for p.

Example 16. Test case t_2 of Figure 7 can be generated from v in Figure 2:

1. Initially, $S = v \,\mathbf{after}\, \epsilon = \{v_0\}$.
2. In the first step input $?but$ is tried: $t_2 := ?but; t_2^2 \,\square\, !liq; \mathbf{fail} \,\square\, !choc; \mathbf{fail}$, after which $S = \{v_0\} \,\mathbf{after}\, ?but = \{v_0, v_1\}$.
3. The allowed outputs are checked: $out(S) = out(\{v_0, v_1\}) = \{!liq, \delta\}$. This leads to the test case $t_2^2 := !liq; t_2^3 \,\square\, !choc; \mathbf{fail} \,\square\, \theta; t_2^4$.
4. For t_2^3 we continue with $S = \{v_0, v_1\} \,\mathbf{after}\, !liq = \{v_0\}$. Another input $?but$ is tried: $t_2^3 := ?but; t_2^5 \,\square\, !liq; \mathbf{fail} \,\square\, !choc; \mathbf{fail}$.
5. Then the output is checked again, which may be $!liq$ or δ: $t_2^5 := !liq; t_2^6 \,\square\, !choc; \mathbf{fail} \,\square\, \theta; t_2^7$.
6. The test case is stopped: $t_2^6 := \mathbf{pass}$ and $t_2^7 := \mathbf{pass}$.
7. Further with t_2^4: this is the test case after quiescence has been observed; t_2^4 is generated from $S = \{v_0, v_1\} \,\mathbf{after}\, \delta = \{v_0\}$. From $\{v_0\}$ another input $?but$ can be supplied: $t_2^4 := ?but; t_2^8 \,\square\, !liq; \mathbf{fail} \,\square\, !choc; \mathbf{fail}$.
8. Analogous to t_2^5 the output is checked: $t_2^8 := !liq; t_2^9 \,\square\, !choc; \mathbf{fail} \,\square\, \theta; t_2^{10}$.
9. After this the test case is stopped: $t_2^9 := \mathbf{pass}$ and $t_2^{10} := \mathbf{pass}$.

When concatenating these pieces the test case t_2 of Figure 7 is obtained. It is clear that this is only one test case which can be generated. Infinitely many different test cases can be generated from specification v by considering longer and longer test cases.

5.3 Completeness of Test Generation

Now all ingredients are there to present the main result of the **ioco** test theory, viz. that the test cases generated with Algorithm 1 can detect all, and only all, non-**ioco** correct implementations. Before giving this completeness result we first formally define what completeness is in the context of **ioco** testing. Moreover, a complete test suite is usually infinitely large, and not executable in a practical situation, as is shown, for instance, for the system v in Example 16. Consequently, a distinction is made between test suites which detect only errors – but possibly not all of them – and test suites which detect all errors – and possibly more. The former are called sound, and the latter exhaustive; see also Section 2.

Definition 17. *Let s be a specification and T a test suite; then for* **ioco***:*

$$\begin{array}{lll}
T \textit{ is complete} & \Leftrightarrow_{\text{def}} & \forall i \in \mathcal{IOTS}(L_I, L_U): \; i \;\mathbf{ioco}\; s \quad \text{iff} \quad i \;\mathbf{passes}\; T \\
T \textit{ is sound} & \Leftrightarrow_{\text{def}} & \forall i \in \mathcal{IOTS}(L_I, L_U): \; i \;\mathbf{ioco}\; s \quad \text{implies} \quad i \;\mathbf{passes}\; T \\
T \textit{ is exhaustive} & \Leftrightarrow_{\text{def}} & \forall i \in \mathcal{IOTS}(L_I, L_U): \; i \;\mathbf{ioco}\; s \quad \text{if} \quad i \;\mathbf{passes}\; T
\end{array}$$

Theorem 2.1 expresses that all tests generated with the algorithm are sound, i.e., give the result **fail** only if the implementation is not **ioco**-correct. Theorem 2.2 states that all possible test cases together form an exhaustive (and thus complete) test suite, which means that for any **ioco**-incorrect implementation there is, in principle, a test case generated with Algorithm 1, that can detect that incorrect implementation.

Theorem 2. *Let $s \in \mathcal{LTS}(L_I, L_U)$ be a specification, and let T_s be the set of all test cases that can be generated from s with algorithm 1; let $gen : \mathcal{LTS}(L_I, L_U) \rightarrow \mathcal{P}(\mathcal{TTS}(L_U, L_I))$ be a test derivation function satisfying $gen(s) \subseteq T_s$; then:*

1. *$gen(s)$ is sound for s with respect to* **ioco***;*
2. *T_s is exhaustive for s with respect to* **ioco***.*

Exhaustiveness is more a theoretical result than a practical one. Theoretically, it implies that any non-conforming implementation can be detected, i.e., that there are no errors that can never be detected. From a practical perspective, an exhaustive test suite, except for the most trivial systems such as p in Figure 2, will contain infinitely many test cases, and thus can never be executed in finite time. The question of which test cases to generate and execute from the infinitely large exhaustive test suite is referred to as *test selection*. Such a selection of test cases should minimize the test costs, e.g., in terms of the necessary test execution time, while maximizing the probability of detecting errors. Test selection is an important yet difficult topic, but it is not further discussed here.

Example 17. In Example 15 test case t_1 of Figure 7 was generated from specification p in Figure 2. Indeed, we had in Example 10: k_1 **ioco** p, k_2 **io$\not$co** p, and k_3 **io$\not$co** p, which is consistent with the test execution results from Example 14: k_1 **passes** t_1, k_2 **fails** t_1, and k_3 **fails** t_1.

5.4 Bibliographic Notes

In the original definition of test cases, and in the original test generation algorithm, test cases were not input enabled [6]. This resulted in a paradox where environments are assumed to be input enabled for the outputs of the system, but test cases, being particular environments, are not. It also meant that test cases could prevent the system from performing an output. Inspired by timed test generation algorithms [23], and by [7], test cases were redefined to be input enabled.

The issue of test selection is studied in many papers, e.g., by using test purposes [2, 35], by using metrics [36, 37], by defining an integral over the space of implementations [38], by approximate analysis [39], by coverage analysis [40], and many others.

An annotated bibliography of testing transition systems can be found in [41].

6 Concluding Remarks

This contribution has presented a model based testing theory for labelled transition systems. Labelled transition systems were introduced as models for specifications, implementations, and tests, and a process language for representing complex transition systems was given. An important point of the theory is the definition of formal correctness between a specification and an implementation. This was done with the implementation relation **ioco**. Some variants of **ioco** were briefly discussed, and in particular the notion of partial specification has been elaborated. A test generation algorithm was given, and it was proved to be complete, i.e., to generate test suites which can exactly test for **ioco** conformance.

Although the emphasis in this contribution was on the theory of model based testing, such theory is only useful if it is supported by model based test tools, in particular test generation tools. Although the principles of the test generation algorithm are not very complex, the application to any realistically sized transition system specification is far beyond what is manually feasible. And, of course, by trying to do it manually, one of the great benefits of model based testing, viz. the automatic generation of large quantities of large tests, would be lost. Prototype tools implementing this test theory exist, e.g., TVEDA [1], TGV [2], THE AGEDIS TOOL SET [3], TESTGEN [4], and TORX [5], and also quite a number of case studies have been performed with these tools, see, e.g., [42–45].

One of the most important open issues in this model based testing theory is the question of test selection. Since exhaustive testing of any realistic system is not an option, an important question is which test cases should be generated and executed, and why one test suite is better than another one. The tools mentioned above use different approaches, from completely random as in TORX, to a manual approach where a user has to provide *test purposes* to steer the selection process. Other approaches are defining some measures of coverage, e.g., using heuristics for coverage of transition systems such as traversing every state at least once, heuristic measures from classical software testing such as equivalence partitioning or boundary value analysis, explicitly defining fault models, or assuming some test hypothesis for the implementation under test. Related to the question of test selection is the issue of how the completeness, coverage, or quality of an automatically generated test suite can be expressed, measured, and, ultimately, controlled. Even more intriguing is the question how a measure of test suite quality can be related to a measure of product quality. After all, product quality is the ultimate reason to make efforts to do testing.

References

1. Phalippou, M.: Relations d'Implantation et Hypothèses de Test sur des Automates à Entrées et Sorties. PhD thesis, L'Université de Bordeaux I, France (1994)

2. 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) (2005) 297–315
3. Hartman, A., Nagin, K.: The AGEDIS Tools for Model Based Testing. In: Int. Symposium on Software Testing and Analysis – ISSTA 2004, New York, USA, ACM Press (2004) 129–132
4. He, J., Turner, K.: Protocol-Inspired Hardware Testing. In Csopaki, G., Dibuz, S., Tarnay, K., eds.: Int. Workshop on Testing of Communicating Systems 12, Kluwer Academic Publishers (1999) 131–147
5. 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 (2003) 13 pages.
6. Tretmans, J.: Test generation with inputs, outputs and repetitive quiescence. Software—Concepts and Tools **17**(3) (1996) 103–120 Also: Technical Report No. 96-26, Centre for Telematics and Information Technology, University of Twente, The Netherlands.
7. Petrenko, A., Yevtushenko, N., Huo, J.L.: Testing Transition Systems with Input and Output Testers. In Hogrefe, D., Wiles, A., eds.: TestCom 2003 – 15^{th} IFIP Int. Conference on Testing of Communicating Systems. Volume 2644 of Lecture Notes in Computer Science., Springer-Verlag (2003) 129–145
8. Brinksma, E., Alderden, R., Langerak, R., Lagemaat, J.v.d., Tretmans, J.: A formal approach to conformance testing. In de Meer, J., Mackert, L., Effelsberg, W., eds.: Second Int. Workshop on Protocol Test Systems, North-Holland (1990) 349–363
9. Tretmans, J.: Testing Concurrent Systems: A Formal Approach. In Baeten, J., Mauw, S., eds.: CONCUR'99 – 10^{th} Int. Conference on Concurrency Theory. Volume 1664 of Lecture Notes in Computer Science., Springer-Verlag (1999) 46–65
10. Bernot, G., Gaudel, M.G., Marre, B.: Software testing based on formal specifications: a theory and a tool. Software Engineering Journal (November) (1991) 387–405
11. Gaudel, M.C.: Testing can be formal, too. In Mosses, P., Nielsen, M., Schwartzbach, M., eds.: TAPSOFT'95: Theory and Practice of Software Development. Volume 915 of Lecture Notes in Computer Science., Springer-Verlag (1995) 82–96
12. ISO/IEC JTC1/SC21 WG7, ITU-T SG 10/Q.8: Information Retrieval, Transfer and Management for OSI – Framework: Formal Methods in Conformance Testing. Committee Draft CD 13245-1, Proposed ITU-T Recommendation Z.500. ISO – ITU-T, Geneve (1997)
13. Petrenko, A.: Fault Model-Driven Test Derivation from Finite State Models: Annotated Bibliography. In Cassez, F., Jard, C., Rozoy, B., Ryan, M., eds.: Modeling and Verification of Parallel Processes – 4^{th} Summer School MOVEP 2000. Volume 2067 of Lecture Notes in Computer Science., Springer-Verlag (2001) 196–205
14. Campbell, C., W., G., Nachmanson, L., Schulte, W., Tillmann, N., Veanes, M.: Model-Based Testing of Object-Oriented Reactive Systems with Spec Explorer. Technical Report MSR-TR-2005-59, Microsoft Research, Redmond, USA (2005)
15. Koopman, P., Alimarine, A., Tretmans, J., Plasmeijer, R.: Gast: Generic Automated Software Testing. In Peña, R., ed.: IFL 2002 – Implementation of Functional Programming Languages. Volume 2670 of Lecture Notes in Computer Science., Springer-Verlag (2003) 84–100
16. Milner, R.: Communication and Concurrency. Prentice-Hall (1989)
17. Bolognesi, T., Brinksma, E.: Introduction to the ISO specification language LOTOS. Computer Networks and ISDN Systems **14** (1987) 25–59

18. ISO: Information Processing Systems, Open Systems Interconnection, LOTOS - A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour. International Standard IS-8807. ISO, Geneve (1989)
19. Lynch, N., Tuttle, M.: An introduction to Input/Output Automata. CWI Quarterly **2**(3) (1989) 219–246 Also: Technical Report MIT/LCS/TM-373 (TM-351 revised), Massachusetts Institute of Technology, Cambridge, U.S.A., 1988.
20. Heerink, L.: Ins and Outs in Refusal Testing. PhD thesis, University of Twente, Enschede, The Netherlands (1998)
21. Krichen, M., Tripakis, S.: Black-Box Conformance Testing for Real-Time Systems. In: 11^{th} Int. SPIN Workshop on Model Checking of Software – SPIN'04. Volume 2989 of Lecture Notes in Computer Science., Springer-Verlag (2004)
22. Larsen, K., Mikucionis, M., Nielsen, B.: Online Testing of Real-Time Systems using UPPAAL. In Grabowski, J., Nielsen, B., eds.: Formal Approaches to Software Testing – FATES 2004. Volume 3395 of Lecture Notes in Computer Science., Springer-Verlag (2005) 79–94
23. Brandán Briones, L., Brinksma, E.: A Test Generation Framework for *quiescent* Real-Time Systems. In Grabowski, J., Nielsen, B., eds.: Formal Approaches to Software Testing – FATES 2004. Volume 3395 of Lecture Notes in Computer Science., Springer-Verlag (2005) 64–78
24. Bijl, M.v.d., Rensink, A., Tretmans, J.: Action Refinement in Conformance Testing. In Khendek, F., Dssouli, R., eds.: TestCom 2005 – 17^{th} IFIP Int. Conference on Testing of Communicating Systems. Volume 3502 of Lecture Notes in Computer Science., Springer-Verlag (2005) 81–96
25. Frantzen, L., Tretmans, J., Willemse, T.: Test Generation Based on Symbolic Specifications. In Grabowski, J., Nielsen, B., eds.: Formal Approaches to Software Testing – FATES 2004. Volume 3395 of Lecture Notes in Computer Science., Springer-Verlag (2005) 1–15
26. De Nicola, R., Hennessy, M.: Testing Equivalences for Processes. Theoretical Computer Science **34** (1984) 83–133
27. De Nicola, R.: Extensional equivalences for transition systems. Theoretical Computer Science **24** (1987) 211–237
28. Phillips, I.: Refusal testing. Theoretical Computer Science **50**(2) (1987) 241–284
29. Langerak, R.: A testing theory for LOTOS using deadlock detection. In Brinksma, E., Scollo, G., Vissers, C.A., eds.: Protocol Specification, Testing, and Verification IX, North-Holland (1990) 87–98
30. Brinksma, E., Scollo, G., Steenbergen, C.: LOTOS specifications, their implementations and their tests. In Bochmann, G.v., Sarikaya, B., eds.: Protocol Specification, Testing, and Verification VI, North-Holland (1987) 349–360
31. Segala, R.: Quiescence, fairness, testing, and the notion of implementation. In Best, E., ed.: CONCUR'93, Lecture Notes in Computer Science 715, Springer-Verlag (1993) 324–338
32. Vaandrager, F.: On the relationship between process algebra and Input/Output Automata. In: Logic in Computer Science, Sixth Annual IEEE Symposium, IEEE Computer Society Press (1991) 387–398
33. Bijl, M.v.d., Rensink, A., Tretmans, J.: Compositional Testing with IOCO. In Petrenko, A., Ulrich, A., eds.: FATES 2003 – Formal Approaches to Software Testing. Volume 2931 of Lecture Notes in Computer Science., Springer-Verlag (2004) 86–100
34. Huo, J.L., Petrenko, A.: On Testing Partially Specified IOTS through Lossless Queues. In Groz, R., Hierons, R., eds.: TestCom 2004 – 16^{th} IFIP Int. Conference

on Testing of Communicating Systems. Volume 2978 of Lecture Notes in Computer Science., Springer-Verlag (2004)
35. Vries, R.d., Tretmans, J.: Towards Formal Test Purposes. In Brinksma, E., Tretmans, J., eds.: Formal Approaches to Testing of Software – FATES'01. Number NS-01-4 in BRICS Notes Series, University of Aarhus, Denmark, BRICS (2001) 61–76
36. 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 10, Chapman & Hall (1997) 200–219
37. 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, Kluwer Academic Publishers (2002) 267–282
38. Brinksma, E.: On the coverage of partial validations. In Nivat, M., Rattray, C., Rus, T., Scollo, G., eds.: AMAST'93, BCS-FACS Workshops in Computing Series, Springer-Verlag (1993) 247–254
39. Jeannet, B., Jéron, T., Rusu, V., Zinovieva, E.: Symbolic Test Selection based on Approximate Analysis. In: Proceedings TACAS'05. Volume 3440 of Lecture Notes in Computer Science., Springer-Verlag (2005)
40. Groz, R., Charles, O., Renévot, J.: Relating Conformance Test Coverage to Formal Specifications. In Gotzhein, R., ed.: FORTE'96, Chapman & Hall (1996)
41. Brinksma, E., Tretmans, J.: Testing Transition Systems: An Annotated Bibliography. In Cassez, F., Jard, C., Rozoy, B., Ryan, M., eds.: Modeling and Verification of Parallel Processes – 4^{th} Summer School MOVEP 2000. Volume 2067 of Lecture Notes in Computer Science., Springer-Verlag (2001) 187–195
42. Groz, R., Risser, N.: Eight Years of Experience in Test Generation from FDTs using TVEDA. In Mizuno, T., Shiratori, N., Higashino, T., Togashi, A., eds.: Formal Desciption Techniques and Protocol Specification, Testing and Verification FORTE X /PSTV XVII '97, Chapman & Hall (1997)
43. Clarke, D., Jéron, T., Rusu, V., Zinovieva, E.: Automated Test and Oracle Generation for Smart-Card Applications. In: E-Smart – Int. Conference on Research in Smart Cards. Volume 2140 of Lecture Notes in Computer Science. (2001) 58–70
44. I., C., Sardis, M., Heuillard, T.: AGEDIS Case Studies: Model-Based Testing in Industry. In Hartman, A., Dussa-Zieger, K., eds.: First European Conference on Model-Driven Software Engineering, Imbuss, Möhrendorf, Germany (2003)
45. Vries, R.d., 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, Kluwer Academic Publishers (2002) 219–234