Jens Grabowski
Brian Nielsen (Eds.)

LNCS 3395

Formal Approaches to Software Testing

4th International Workshop, FATES 2004
Linz, Austria, September 2004
Revised Selected Papers

Springer

TP311.5-53
F723
2004

Jens Grabowski Brian Nielsen (Eds.)

Formal Approaches to Software Testing

4th International Workshop, FATES 2004
Linz, Austria, September 21, 2004
Revised Selected Papers

Springer

Volume Editors

Jens Grabowski
University of Göttingen
Institute for Informatics
Lotzestr. 16-18, 37083 Göttingen, Germany
E-mail: grabowski@informatik.uni-goettingen.de

Brian Nielsen
Aalborg University
Department of Computer Science
Fredrik Bajersvej 7B, 9220 Aalborg, Denmark
E-mail: bnielsen@cs.auc.dk

Library of Congress Control Number: 2005921470

CR Subject Classification (1998): D.2, D.3, F.3, K.6

ISSN 0302-9743
ISBN 3-540-25109-X Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 11400752 06/3142 5 4 3 2 1 0

Preface

Testing often accounts for more than 50% of the required effort during system development. The challenge for research is to reduce these costs by providing new methods for the specification and generation of high-quality tests. Experience has shown that the use of formal methods in testing represents a very important means for improving the testing process. Formal methods allow for the analysis and interpretation of models in a rigorous and precise mathematical manner. The use of formal methods is not restricted to system models only. Test models may also be examined. Analyzing system models provides the possibility of generating complete test suites in a systematic and possibly automated manner whereas examining test models allows for the detection of design errors in test suites and their optimization with respect to readability or compilation and execution time. Due to the numerous possibilities for their application, formal methods have become more and more popular in recent years.

The Formal Approaches in Software Testing (FATES) workshop series also benefits from the growing popularity of formal methods. After the workshops in Aalborg (Denmark, 2001), Brno (Czech Republic, 2002) and Montréal (Canada, 2003), FATES 2004 in Linz (Austria) was the fourth workshop of this series. Similar to the workshop in 2003, FATES 2004 was organized in affiliation with the IEEE/ACM Conference on Automated Software Engineering (ASE 2004). FATES 2004 received 41 submissions. Each submission was reviewed by at least three independent reviewers from the Program Committee with the help of some additional reviewers. Based on their evaluations, 14 full papers and one work-in-progress paper from 11 different countries were selected for presentation.

This volume contains revised versions of the presented papers. The revisions reflect the lively discussions among the presenters and participants during the FATES workshop. The papers use different formal methods and languages, e.g., automata, labelled transition systems, TTCN-3 or UPPAAL, and apply them to symbolic test generation, the use of model-checking techniques in testing, the test of nonfunctional properties, and test optimization. This diversity of formal methods and application domains in conjunction with the high number of submissions to and participants of the FATES 2004 workshop emphasize the increased importance attributed to the research on formal approaches in software testing.

We would like to express our gratitude to all authors for their valuable contributions and to the Workshop Organizing Committee of the ASE 2004 conference. In addition, we would like to thank all members of the FATES Program Committee and the additional reviewers, who were given the essential task of reviewing many papers in a short period of time. The individuals who contributed to this effort are listed on the following pages.

December 2004
Goettingen and Aalborg

Jens Grabowski and Brian Nielsen
Program Chairs
FATES 2004

Organization

Program Chairs

Jens Grabowski	University of Goettingen, Germany
Brian Nielsen	Aalborg University, Denmark

Program Committee

Rachel Cardell-Oliver	University of Western Australia, Crawley, Australia
Shing-Chi Cheung	Hong Kong University of Science and Technology, Hong Kong, China
Marie-Claude Gaudel	Université de Paris-Sud, France
Wolfgang Grieskamp	Microsoft Research, USA
Robert M. Hierons	Brunel University, UK
Thierry Jron	IRISA/INRIA, France
David Lee	Bell Labs, Beijing, China
Jose Carlos Maldonado	University of Sao Paulo, Brazil
Manuel Nunez	Universidad Complutense de Madrid, Spain
Jeff Offutt	George Mason University, USA
Alexandre Petrenko	Computer Research Institute of Montréal, Canada
Ina Schieferdecker	Fraunhofer FOKUS, Berlin, Germany
Jan Tretmans	Radboud University, Nijmegen, The Netherlands
Andreas Ulrich	Siemens AG, Corporate Technology, Munich, Germany
Carsten Weise	Ericsson Eurolab Deutschland GmbH, Germany
Clay Williams	IBM Research, Thomas J. Watson Research Center, New York, USA

Additional Reviewers

Aynur Abdurazik	George Mason University, USA
Roger Alexander	Colorado State University, USA
Ellen Francine Barbosa	University of Sao Paulo, Brazil
Machiel van der Bijl	University of Twente, The Netherlands
Henrik Bohnenkamp	University of Twente, The Netherlands
Sergiy Boroday	CRIM, Canada
Ricky W.K. Chan	University of Hong Kong, Hong Kong, China
Caixia Chi	Bell Labs, China
Lars Frantzen	Radboud University, Nijmegen, The Netherlands
Tim French	University of Western Australia, Australia
David de Frutos-Escrig	Universidad Complutense de Madrid, Spain
Roland Groz	INPG-ENSIMAG, Canada
Yuri Gurevich	Microsoft Research, Redmond, USA
Cedric S.C. Ho	Hong Kong University of Science and Technology, Hong Kong, China
Jin Bei-Hong	Chinese Academy of Sciences, China
Jia Le Huo	CRIM, Canada
Matthew Kaplan	IBM Research, Thomas J. Watson Research Center, New York, USA
Tim Klinge	IBM Research, Thomas J. Watson Research Center, New York, USA
Pieter Koopman	Radboud University, Nijmegen, The Netherlands
Keqin Li	Bell Labs, China
Zhijun Liu	Ohio State University, USA
Marius Mikucionis	Aalborg University, Denmark
Helmut Neukirchen	University of Goettingen, Germany
Vikram Reddy	Ohio State University, USA
Ismael Rodrguez	Universidad Complutense de Madrid, Spain
Fernando Rubio	Universidad Complutense de Madrid, Spain
Guoqiang Shu	Ohio State University, USA
Arne Skou	Aalborg University, Denmark
Tatiana Sugeta	University of Sao Paulo, Brazil
Nikolai Tillmann	Microsoft Research, Redmond, USA
Margus Veanes	Microsoft Research, Redmond, USA
Auri M.R. Vincenzi	University of Sao Paulo, Brazil
Bijendra Vishal	Ohio State University, USA
Frdric Voisin	LRI, Université de Paris-Sud and CNRS, France
Dong Wang	Bell Labs, China
Edith Werner	University of Goettingen, Germany
Tim Willemse	Radboud University Nijmegen, The Netherlands

Table of Contents

Test Optimization

Test Generation Based on Symbolic Specifications

Lars Frantzen[*], Jan Tretmans, and Tim A.C. Willemse[**]

Nijmegen Institute for Computing and Information Sciences (NIII),
Radboud University Nijmegen – The Netherlands
`{lf, tretmans, timw}@cs.ru.nl`

Abstract. Classical state-oriented testing approaches are based on simple machine models such as Labelled Transition Systems (LTSs), in which data is represented by concrete values. To implement these theories, data types which have infinite universes have to be cut down to finite variants, which are subsequently enumerated to fit in the model. This leads to an explosion of the state space. Moreover, exploiting the syntactical and/or semantical information of the involved data types is non-trivial after enumeration. To overcome these problems, we lift the family of testing relations $\mathbf{ioco}_{\mathcal{F}}$ to the level of Symbolic Transition Systems (STSs). We present an algorithm based on STSs, which generates and executes tests on-the-fly on a given system. It is sound and complete for the $\mathbf{ioco}_{\mathcal{F}}$ testing relations.

1 Introduction

Testing is an important technique to assess the quality of systems. In testing, experiments are conducted with a System Under Test (SUT) to determine whether it behaves as expected. There are many different kinds of testing. We focus on formal, specification based, black box, functionality testing. This basically means that the SUT can only be observed (and controlled) via its external interfaces. Moreover, a mathematical, unambiguous specification of the causal order between (appropriate) inputs and expected outputs of the SUT is the starting point for the generation and the analysis of the test results.

Several (formal) test generation tools have been developed for specification based, black box testing. Most of these tools use (variations of) state machines or transition systems as the underlying model for test generation. We refer to these types of tools as *state oriented* tools. For an overview of such tools see [2]. A problem, often encountered in such tools is the *state space explosion*, which is

[*] Lars Frantzen is supported by the Netherlands Organisation for Scientific Research (NWO) under project: STRESS – Systematic Testing of Realtime Embedded Software Systems.

[**] Tim Willemse carried out this work as part of the TANGRAM project under the responsibility of the Embedded Systems Institute. Tangram is partially supported by the Netherlands Ministry of Economic Affairs under grant TSIT2026.

due to the fact that they use an explicit internal representation for the states of the specification. This is particularly true when the specification uses complex data structures with large or infinite data domains, because each value in the data domain potentially leads to another state. Consequently, many tools can only cope with very restricted data structures with finite domains.

Opposed to state oriented tools are *data type oriented* tools, which are tools tailored to deal with test generation for complicated data structures, such as QuickCheck [3] and Gast [5]. These tools employ the structure of data types to generate test data. However, they lack a built-in concept of state, which makes them less suited to test, e.g., concurrent systems. The way to handle state in such tools is to explicitly define a data structure that represents a state space, but this is not always satisfactory.

The combination of the state oriented and the data type oriented approaches looks promising, and it is exactly this what we investigate in this paper. As our basis we take a state oriented approach to testing, viz. the **ioco** test theory [8]. To the underlying model of Labelled Transition Systems, we add the concept of location variables, and the concept of data, which can be communicated over gates. Both influence the flow of control, thereby allowing us to specify data-dependent behaviour. We refer to these augmented Labelled Transition Systems as *Symbolic Transition Systems* (STSs). We subsequently lift the **ioco** test theory to STSs. As a result, we obtain a sound and complete test derivation algorithm from specifications expressed as STSs.

The test derivation algorithm for STSs allows to treat data symbolically. Rather than elaborating our approach for a specific data formalism, data types are treated as sets of values (algebras) and first order formulas are used to specify values or predicates. This allows to combine STSs with any formalism of choice (with corresponding test tools) for the specification and manipulation of data. This is further elaborated into a tractable algorithm.

From a theoretical point of view, it is also interesting to give an algorithm which generates *symbolic test cases* (STCs). This requires a purely symbolic version of the $\mathbf{ioco}_{\mathcal{F}}$ relations. This is depicted in Fig. 1. The front triangle

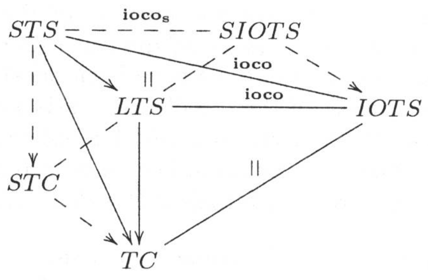

Fig. 1. Classical **ioco** test theory and symbolic **ioco** test theory

represents the classical **ioco** test theory, as presented in [8]. Test cases (TC) are generated out of a specification LTS, and subsequently executed ($||$) on an SUT, assumed to be modelled by an IOTS. The rear triangle consists of a purely symbolic test theory. In this paper, we concentrate on the relation between STSs, LTSs and IOTSs, and on the generation and execution of test cases, i.e. the relation between STSs and TCs. Elaborating on the dashed lines and the corresponding models is another line of research we are pursuing.

Related Work. The idea of combining data type oriented and state oriented approaches is not entirely new in testing. We mention a few noteworthy approaches.

The approach which comes closest to ours is the one described in [7]. There, Input-Output Symbolic Transition Systems (IOSTSs) are used, which are very similar to our STSs. The conformance relation they use corresponds to **ioconf** = $\mathbf{ioco}_{traces(\mathcal{L})}$, but they do not deal with quiescence. In [7] test purposes are chosen as a way to tackle the state space explosion problem. These are used to compute a subgraph of the IOSTS representing a specific issue of interest. Such test purposes are again (special) IOSTSs. The result is a test case which is still symbolic in the sense that it is a deterministic IOSTS with special states *Pass*, *Fail* and *Inconclusive*. The verdict *Inconclusive* is necessary to judge a behaviour which conforms to a given specification, but does not satisfy the given test purpose. Our approach does not rely on test purposes, even though the set $\mathcal{F}$ which identifies the relation $\mathbf{ioco}_{\mathcal{F}}$ can be seen as some form of test purpose.

The data-type oriented GAST tool [5] was recently extended in [6] to deal with specifications given as (possibly nondeterministic) Extended Finite State Machines (EFSMs). Such EFSMs are also symbolic specifications, but in some senses more restrictive than STSs or IOSTSs. GAST basically implements a generic algorithm to enumerate the elements of an arbitrary algebraic data type. Such a type can be an input value, but also a whole path through the EFSM. Since the list of all elements of a recursive type is infinitely long, lazy evaluation is employed to generate only the fraction of this list that is actually needed. The elements are generated in increasing size, both the executed paths and the input values. GAST can be used to execute the generated tests on an SUT in an on-the-fly manner.

Overview. This paper is structured as follows. In Sect. 2 we briefly repeat notions from first order logic. The **ioco** test theory is summarised in Sect. 3. The framework of Symbolic Transition Systems is introduced in Sect. 4. We present an on-the-fly implementation for generating and executing test cases for Symbolic Transition Systems in Sect. 5. We finish with conclusions and future extensions in Sect. 6.

2 First Order Logic

We use basic concepts from first order logic as our framework for dealing with data. For a general introduction into logic we refer to [4]. From hereon we assume a first order structure as given, i.e.:

- A logical signature $\mathfrak{S} = (F,\ P)$ with
  - F is a set of *function symbols.* Each $f{\in}F$ has a corresponding arity $n{\in}\mathbb{N}$. If $n = 0$ we call f a *constant.*
  - P is a set of *predicate symbols.* Each $p{\in}P$ has a corresponding arity $n{>}0$.
- A model $\mathfrak{M} = (\mathfrak{U},\ (f_{\mathfrak{M}})_{f\in F},\ (p_{\mathfrak{M}})_{p\in P})$ with
  - $\mathfrak{U}$ being a nonempty set called *universe.*
  - For all $f{\in}F$ with arity n, $f_{\mathfrak{M}}$ is a function of type $\mathfrak{U}^n{\rightarrow}\mathfrak{U}$.
  - For every $p{\in}P$ with arity n we have $p_{\mathfrak{M}} \subseteq \mathfrak{U}^n$.

For simplicity, and without loss of generality we restrict to one-sorted signatures. Let $\mathfrak{X}$ be a set of *variables. Terms* over X, denoted $\mathfrak{T}(X)$, are built from function symbols F and variables $X \subseteq \mathfrak{X}$. We write $\mathsf{var}(t)$ to denote the set of variables appearing in a term t. Terms $t{\in}\mathfrak{T}(\emptyset)$ are called *ground terms.*

Example 1. Assume we have $X = \{x, y\}$. Let $\mathfrak{S} = (F,\ P)$ be given by $F = \{\texttt{zero}, \texttt{succ}, \texttt{add}\}$ (with arities $0, 1$ and 2, resp.), and $P = \{\texttt{leq}\}$ (with arity 2). An obvious model for this signature is the natural numbers with 0, *successor, addition* and the less-or-equal predicate; any other model that sticks to the given arities is fine too. Terms are, e.g. x, $\texttt{succ}(x)$ and $\texttt{add}(\texttt{succ}(x), y)$. Ground terms are, e.g. $\texttt{zero}$ and $\texttt{add}(\texttt{zero}, \texttt{succ}(\texttt{zero}))$. □

A *term-mapping* is a function $\sigma{:}\mathfrak{X} \rightarrow \mathfrak{T}(\mathfrak{X})$. The term-mapping id, referred to as the *identity mapping,* is defined as $\mathsf{id}(x) = x$ for all $x{\in}\mathfrak{X}$. We use the following notation. For sets X, Y with $X \cup Y \subseteq \mathfrak{X}$, we write $\mathfrak{T}(Y)^X$ for the set of term-mappings that assign to each variable $x{\in}X$ a term $t{\in}\mathfrak{T}(Y)$, and to each variable $x \notin X$ the term x. Given a term-mapping $\sigma{\in}\mathfrak{T}(Y)^X$ we overload the var-notation as follows: $\mathsf{var}(\sigma) =_{def} \bigcup_{x\in X} \mathsf{var}(\sigma(x))$.

The set of free variables of a first order formula φ is denoted $\mathsf{free}(\varphi)$; the set of bound variables is denoted $\mathsf{bound}(\varphi)$. The set of first order formulas φ over $X \subseteq \mathfrak{X}$ is denoted $\mathfrak{F}(X)$; we have $\mathsf{free}(\varphi) \cup \mathsf{bound}(\varphi) \subseteq X$. A tautology is represented by $\top$. The *existential closure* of a formula φ, denoted $\overline{\exists}\varphi$, is defined as $\overline{\exists}\varphi =_{def} \exists x_1 \exists x_2 \ldots \exists x_n : \varphi$ with $\{x_1, \ldots, x_n\} = \mathsf{free}(\varphi)$.

Given a term-mapping σ and a formula φ, the *substitution* of $\sigma(x)$ for $x{\in}\mathsf{free}(\varphi)$ in φ is denoted $\varphi[\sigma]$. Substitutions are side-effect free, i.e. they do not add bound variables. This is achieved using α-renaming. The substitution of terms $\sigma(x)$ for variables $x{\in}\,\mathsf{var}(t)$, in a term t using a term-mapping σ, is denoted $t[\sigma]$.

Example 2. An example of a term mapping for $X = \{x, y\}$ is $\sigma = \{x \mapsto \texttt{succ}(y),\ y \mapsto \texttt{zero}\}{\in}\mathfrak{T}(X)^X$, with $\mathsf{var}(\sigma) = \{y\}$. The existential closure of the formula $\varphi = \forall y : \texttt{leq}(x, y)$ with $\mathsf{bound}(\varphi) = \{y\}$ and $\mathsf{free}(\varphi) = \{x\}$ is $\overline{\exists}\varphi = \exists x \forall y : \texttt{leq}(x, y)$. The substitution of σ in φ is not side-effect free, but can be achieved by renaming variable y to z, i.e. $\varphi[\sigma] = \forall z : \texttt{leq}(\texttt{succ}(y), z)$. □

A *valuation* ϑ is a function $\vartheta{:}\mathfrak{X} \rightarrow \mathfrak{U}$. We denote the set of all valuations as $\mathfrak{U}^{\mathfrak{X}} =_{def} \{\vartheta{:}\mathfrak{X} \rightarrow \mathfrak{U} \mid \vartheta \text{ is a valuation of } \mathfrak{X}\}$. For a given $X \subseteq \mathfrak{X}$ we write $\vartheta{\in}\mathfrak{U}^X$ when only the values of the variables in X are of interest. For all the other variables $y{\in}\mathfrak{X} \setminus X$ we set $\vartheta(y) = *$, where $*$ is an arbitrary element of set $\mathfrak{U}$.

Having two valuations $\vartheta \in \mathfrak{U}^X$ and $\varsigma \in \mathfrak{U}^Y$ with $X \cap Y = \emptyset$, their union is defined as:

$$(\vartheta \cup \varsigma)(x) =_{def} \begin{cases} \vartheta(x) \text{ if } x \in X \\ \varsigma(x) \text{ if } x \in Y \\ * \quad \text{otherwise} \end{cases}$$

The *satisfaction* of a formula φ w.r.t. a given valuation ϑ is denoted $\vartheta \models \varphi$. When $\mathsf{free}(\varphi) = \emptyset$ we write $\mathfrak{M} \models \psi$ because the satisfaction is independent of a concrete valuation.

The extension to evaluate whole terms based on a valuation ϑ is called a *term-evaluation* and denoted $\vartheta_{\mathsf{eval}}:\mathfrak{T}(\mathfrak{X}) \rightarrow \mathfrak{U}$. The evaluation of ground terms is denoted $\mathsf{eval}:\mathfrak{T}(\emptyset) \rightarrow \mathfrak{U}$.

To ease notation, we often treat a tuple $\langle x_1, \ldots, x_n \rangle \in A_1 \times \cdots \times A_n$ as the set $\{x_1, \ldots, x_n\}$. We denote the composition of functions $f{:}B \rightarrow C$ and $g{:}A \rightarrow B$ as $f \circ g$.

Example 3. Assuming the standard model for natural numbers as given in example 1, an example valuation is $\vartheta = \{x \mapsto 24,\ y \mapsto 7\} \in \mathfrak{U}^{\{x,y\}}$. For the formula φ of example 2, the valuation ϑ and the standard model for natural numbers we find $\vartheta \not\models \varphi$ and $\mathfrak{M} \models \overline{\exists}\varphi$ and we get $\vartheta_{\mathsf{eval}}(\mathsf{add}(x, \mathsf{succ}(y))) = 32$. □

Our example of a logical structure for natural numbers shows that many, even infinite ground terms may evaluate to the same value, e.g. the ground terms `zero` and `add(zero, zero)` both evaluate to 0. We assume we have a unique ground term representative for every value to facilitate the bidirectional translation.

3 Testing Labelled Transition Systems

We briefly review the **ioco**$_{\mathcal{F}}$ test theory on which this paper is based. For a more detailed overview, we refer to [8]. The semantical model we use to model reactive systems is based on *Labelled Transition Systems* (LTSs).

Definition 1. *A* Labelled Transition System *is a tuple* $\mathcal{L} = \langle S, s_0, \Sigma, \rightarrow \rangle$, *where*

- S *is a (possibly infinite) set of* states.
- $s_0 \in S$ *is the* initial state.
- Σ *is a (possibly infinite) set of* action labels. *The special action label* $\tau \notin \Sigma$ *denotes an* unobservable *action. In contrast, all other actions are* observable. *We write* Σ_τ *to denote the set* $\Sigma \cup \{\tau\}$.
- $\rightarrow \subseteq S \times \Sigma_\tau \times S$ *is the* transition relation. *When* $(s, \mu, s') \in \rightarrow$ *we write* $s \xrightarrow{\mu} s'$.

We often identify an LTS $\mathcal{L}$ *with its initial state* s_0.

Unobservable actions can be used to model events that cannot be seen by an observer of a system. The generalised transition relation $\Longrightarrow \subseteq S \times \Sigma^* \times S$ captures this phenomenon: it abstracts from τ actions preceding, in-between and following a (possibly empty) sequence of observable actions. Given an LTS

Table 1. Deduction rules for generalised transitions

$$s \overset{\epsilon}{\Longrightarrow} s \qquad \frac{s \overset{\sigma}{\Longrightarrow} s'' \quad s'' \xrightarrow{\tau} s'}{s \overset{\sigma}{\Longrightarrow} s'} \qquad \frac{s \overset{\sigma}{\Longrightarrow} s'' \quad s'' \xrightarrow{\mu} s' \quad \mu \neq \tau}{s \overset{\sigma\mu}{\Longrightarrow} s'}$$

$\mathcal{L} = \langle S, s_0, \Sigma, \rightarrow \rangle$, this relation is defined by the deduction rules of Table 1. We define two operations on LTSs. Given an LTS $\mathcal{L} = \langle S, s_0, \Sigma, \rightarrow \rangle$ and a (possibly new) action μ. The *action prefix* $\mu; \mathcal{L}$ is defined as

$$\mu; \mathcal{L} =_{def} \langle S \cup \{s\}, s, \Sigma \cup \{\mu\}, \rightarrow \cup \{s \xrightarrow{\mu} s_0\} \rangle \tag{1}$$

with $s \notin S$ being a fresh state. For a set of LTSs $\overline{\mathcal{L}} = \{\mathcal{L}_1, \ldots, \mathcal{L}_n\}$ with $n \geq 0$ of the form $\mathcal{L}_i = \langle S_i, s_{0i}, \Sigma_i, \rightarrow_i \rangle$, we define the *alternative composition* of all LTSs $\mathcal{L}_i$, denoted $\sum(\overline{\mathcal{L}})$, as follows:

$$\sum(\overline{\mathcal{L}}) =_{def} \langle \bigcup_{i \leq n} S_i \cup \{s\}, s, \bigcup_{i \leq n} \Sigma_i, \bigcup_{i \leq n} (\rightarrow_i \cup \{s \xrightarrow{\mu} s' \mid s_{0i} \xrightarrow{\mu} s'\}) \rangle \tag{2}$$

with $s \notin \bigcup_{i \leq n} S_i$ being a fresh state. The operator $\sum$ is associative and commutative. We sometimes write $\mathcal{L}_1 + \mathcal{L}_2$ instead of $\sum\{\mathcal{L}_1, \mathcal{L}_2\}$.

3.1 The Test Relation $\mathbf{ioco}_{\mathcal{F}}$

We introduce the following shorthand notation. For a $\mu \in \Sigma_\tau$ we write $s \xrightarrow{\mu}$ when there is a state s' such that $s \xrightarrow{\mu} s'$, and, likewise, given a $\sigma \in \Sigma^*$ we write $s \overset{\sigma}{\Longrightarrow}$ when there is a state s' such that $s \overset{\sigma}{\Longrightarrow} s'$.

Definition 2. *Let $\mathcal{L} = \langle S, s_0, \Sigma, \rightarrow \rangle$ be an LTS and let $s \in S$.*

1. *$init(s) =_{def} \{ \mu \in \Sigma_\tau \mid s \xrightarrow{\mu} \}$.*
2. *$traces(s) =_{def} \{ \sigma \in \Sigma^* \mid s \overset{\sigma}{\Longrightarrow} \}$.*
3. *$\mathcal{L}$ has* finite behaviour *if all $\sigma \in traces(s_0)$ satisfy $|\sigma| < n$ for some $n \in \mathbb{N}$.*
4. *$\mathcal{L}$ is* deterministic *if for all $\sigma \in \Sigma^*$, $|\{s' \mid s_0 \overset{\sigma}{\Longrightarrow} s'\}| \leq 1$.*

We assume that implementations of a reactive system can be given as an *input-output transition system* (IOTSs). An IOTS is an LTS in which the set of action labels Σ is partitioned in a set of *input actions* Σ_I and a set of *output actions* Σ_U, and for which it is assumed that all input actions are enabled in all states.

Definition 3. *Let $\mathcal{L} = \langle S, s_0, \Sigma_I \cup \Sigma_U, \rightarrow \rangle$ be an LTS. A state $s \in S$ is* quiescent, *denoted by $\delta(s)$, if $\forall \mu \in \Sigma_U \cup \{\tau\} : s \overset{\mu}{\not\rightarrow}$.*

Let δ be a special action label, not part of any action label set. For a given set of action labels Σ, we abbreviate $\Sigma \cup \{\delta\}$ with Σ_δ. The suspension transitions $\Longrightarrow_\delta \subseteq S \times \Sigma_\delta^* \times S$ are given by the deduction rules of Table 2. The set of all *suspension traces* of $\mathcal{L}$ is denoted $Straces(\mathcal{L}) = \{\sigma \in \Sigma_\delta^* \mid \mathcal{L} \overset{\sigma}{\Longrightarrow}_\delta\}$.