Frank S. de Boer Marcello M. Bonsangue
Susanne Graf Willem-Paul de Roever (Eds.)

Formal Methods for Components and Objects

LNCS 4709

5th International Symposium, FMCO 2006
Amsterdam, The Netherlands, November 2006
Revised Lectures

Springer

Frank S. de Boer Marcello M. Bonsangue
Susanne Graf Willem-Paul de Roever (Eds.)

Formal Methods for Components and Objects

5th International Symposium, FMCO 2006
Amsterdam, The Netherlands, November 7-10, 2006
Revised Lectures

Springer

Volume Editors

Frank S. de Boer
Centre for Mathematics and Computer Science, CWI
Kruislaan 413, 1098 SJ Amsterdam, The Netherlands
E-mail: F.S.de.Boer@cwi.nl

Marcello M. Bonsangue
Leiden University
Leiden Institute of Advanced Computer Science
2300 RA Leiden, The Netherlands
E-mail: marcello@liacs.nl

Susanne Graf
VERIMAG
2 Avenue de Vignate, 38610 Grenoble-Gières, France
E-mail: Susanne.Graf@imag.fr

Willem-Paul de Roever
University of Kiel
Institute of Computer Science and Applied Mathematics
Hermann-Rodewald-Str. 3, 24118 Kiel, Germany
E-mail: wpr@informatik.uni-kiel.de

Library of Congress Control Number: Applied for

CR Subject Classification (1998): D.2, D.3, F.3, D.4

LNCS Sublibrary: SL 2 – Programming and Software Engineering

ISSN	0302-9743
ISBN-10	3-540-74791-5 Springer Berlin Heidelberg New York
ISBN-13	978-3-540-74791-8 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 12119530 06/3180 5 4 3 2 1 0

Lecture Notes in Computer Science 4709

Commenced Publication in 1973
Founding and Former Series Editors:
Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

Preface

Large and complex software systems provide the necessary infrastructure in all industries today. In order to construct such large systems in a systematic manner, the focus in the development methodologies has switched in the last two decades from functional issues to structural issues: both data and functions are encapsulated into software units which are integrated into large systems by means of various techniques supporting reusability and modifiability. This encapsulation principle is essential to both the object-oriented and the more recent component-based software engineering paradigms.

Formal methods have been applied successfully to the verification of medium-sized programs in protocol and hardware design. However, their application to the development of large systems requires more emphasis on specification, modeling and validation techniques supporting the concepts of reusability and modifiability, and their implementation in new extensions of existing programming languages like Java.

The fifth international symposium on Formal Methods for Components and Objects (FMCO 2006) was held in Amsterdam, The Netherlands, June 7–11, 2007. The program consisted of invited keynote lectures and tutorial lectures selected through a corresponding open-call. The latter provide a tutorial perspective on recent developments. In contrast to many existing conferences, about half of the program consisted of invited keynote lectures by top researchers sharing their interest in the application or development of formal methods for large-scale software systems (object or component oriented). FMCO does not focus on specific aspects of the use of formal methods, but rather it aims at a systematic and comprehensive account of the expanding body of knowledge on modern software systems.

This volume contains the contributions submitted after the symposium by both the invited and selected lecturers. The proceedings of FMCO 2002, FMCO 2003, FMCO 2004 and FMCO 2005 have already been published as volumes 2852, 3188, 3657, and 4111 of Springer's *Lecture Notes in Computer Science*. We believe that these proceedings provide a unique combination of ideas on software engineering and formal methods which reflect the expanding body of knowledge on modern software systems.

Finally, we thank all authors for the high quality of their contributions, and the reviewers for their help in improving the papers for this volume.

June 2007

Frank de Boer
Marcello Bonsangue
Susanne Graf
Willem-Paul de Roever

Organization

The FMCO symposia are organized in the context of the project Mobi-J, a project founded by a bilateral research program of The Dutch Organization for Scientific Research (NWO) and the Central Public Funding Organization for Academic Research in Germany (DFG). The partners of the Mobi-J projects are: the Centrum voor Wiskunde en Informatica, the Leiden Institute of Advanced Computer Science, and the Christian-Albrechts-Universität Kiel.

This project aims at the development of a programming environment which supports component-based design and verification of Java programs annotated with assertions. The overall approach is based on an extension of the Java language with a notion of component that provides for the encapsulation of its internal processing of data and composition in a network by means of mobile asynchronous channels.

Sponsoring Institutions

The Dutch Organization for Scientific Research (NWO)
The Royal Netherlands Academy of Arts and Sciences (KNAW)
The Dutch Institute for Programming research and Algorithmics (IPA)
The Centrum voor Wiskunde en Informatica (CWI), The Netherlands
The Leiden Institute of Advanced Computer Science (LIACS), The Netherlands

Lecture Notes in Computer Science

Sublibrary 2: Programming and Software Engineering

For information about Vols. 1– 4257
please contact your bookseller or Springer

Vol. 4902: P. Hudak, D.S. Warren (Eds.), Practical Aspects of Declarative Languages. X, 333 pages. 2007.

Vol. 4888: F. Kordon, O. Sokolsky (Eds.), Composition of Embedded Systems. XII, 221 pages. 2007.

Vol. 4849: M. Winckler, H. Johnson, P. Palanque (Eds.), Task Models and Diagrams for User Interface Design. XIII, 299 pages. 2007.

Vol. 4839: O. Sokolsky, S. Tasiran (Eds.), Runtime Verification. VI, 215 pages. 2007.

Vol. 4834: R. Cerqueira, R.H. Campbell (Eds.), Middleware 2007. XIII, 451 pages. 2007.

Vol. 4829: M. Lumpe, W. Vanderperren (Eds.), Software Composition. VIII, 281 pages. 2007.

Vol. 4824: A. Paschke, Y. Biletskiy (Eds.), Advances in Rule Interchange and Applications. XIII, 243 pages. 2007.

Vol. 4807: Z. Shao (Ed.), Programming Languages and Systems. XI, 431 pages. 2007.

Vol. 4799: A. Holzinger (Ed.), HCI and Usability for Medicine and Health Care. XVI, 458 pages. 2007.

Vol. 4789: M. Butler, M.G. Hinchey, M.M. Larrondo-Petrie (Eds.), Formal Methods and Software Engineering. VIII, 387 pages. 2007.

Vol. 4767: F. Arbab, M. Sirjani (Eds.), International Symposium on Fundamentals of Software Engineering. XIII, 450 pages. 2007.

Vol. 4765: A. Moreira, J. Grundy (Eds.), Early Aspects: Current Challenges and Future Directions. X, 199 pages. 2007.

Vol. 4764: P. Abrahamsson, N. Baddoo, T. Margaria, R. Messnarz (Eds.), Software Process Improvement. XI, 225 pages. 2007.

Vol. 4762: K.S. Namjoshi, T. Yoneda, T. Higashino, Y. Okamura (Eds.), Automated Technology for Verification and Analysis. XIV, 566 pages. 2007.

Vol. 4758: F. Oquendo (Ed.), Software Architecture. XVI, 340 pages. 2007.

Vol. 4757: F. Cappello, T. Herault, J. Dongarra (Eds.), Recent Advances in Parallel Virtual Machine and Message Passing Interface. XVI, 396 pages. 2007.

Vol. 4753: E. Duval, R. Klamma, M. Wolpers (Eds.), Creating New Learning Experiences on a Global Scale. XII, 518 pages. 2007.

Vol. 4749: B.J. Krämer, K.-J. Lin, P. Narasimhan (Eds.), Service-Oriented Computing – ICSOC 2007. XIX, 629 pages. 2007.

Vol. 4748: K. Wolter (Ed.), Formal Methods and Stochastic Models for Performance Evaluation. X, 301 pages. 2007.

Vol. 4741: C. Bessière (Ed.), Principles and Practice of Constraint Programming – CP 2007. XV, 890 pages. 2007.

Vol. 4735: G. Engels, B. Opdyke, D.C. Schmidt, F. Weil (Eds.), Model Driven Engineering Languages and Systems. XV, 698 pages. 2007.

Vol. 4716: B. Meyer, M. Joseph (Eds.), Software Engineering Approaches for Offshore and Outsourced Development. X, 201 pages. 2007.

Vol. 4709: F.S. de Boer, M.M. Bonsangue, S. Graf, W.-P. de Roever (Eds.), Formal Methods for Components and Objects. VIII, 297 pages. 2007.

Vol. 4680: F. Saglietti, N. Oster (Eds.), Computer Safety, Reliability, and Security. XV, 548 pages. 2007.

Vol. 4670: V. Dahl, I. Niemelä (Eds.), Logic Programming. XII, 470 pages. 2007.

Vol. 4652: D. Georgakopoulos, N. Ritter, B. Benatallah, C. Zirpins, G. Feuerlicht, M. Schoenherr, H.R. Motahari-Nezhad (Eds.), Service-Oriented Computing ICSOC 2006. XVI, 201 pages. 2007.

Vol. 4640: A. Rashid, M. Aksit (Eds.), Transactions on Aspect-Oriented Software Development IV. IX, 191 pages. 2007.

Vol. 4634: H. Riis Nielson, G. Filé (Eds.), Static Analysis. XI, 469 pages. 2007.

Vol. 4620: A. Rashid, M. Aksit (Eds.), Transactions on Aspect-Oriented Software Development III. IX, 201 pages. 2007.

Vol. 4615: R. de Lemos, C. Gacek, A. Romanovsky (Eds.), Architecting Dependable Systems IV. XIV, 435 pages. 2007.

Vol. 4610: B. Xiao, L.T. Yang, J. Ma, C. Muller-Schloer, Y. Hua (Eds.), Autonomic and Trusted Computing. XVIII, 571 pages. 2007.

Vol. 4609: E. Ernst (Ed.), ECOOP 2007 – Object-Oriented Programming. XIII, 625 pages. 2007.

Vol. 4608: H.W. Schmidt, I. Crnković, G.T. Heineman, J.A. Stafford (Eds.), Component-Based Software Engineering. XII, 283 pages. 2007.

Vol. 4591: J. Davies, J. Gibbons (Eds.), Integrated Formal Methods. IX, 660 pages. 2007.

Vol. 4589: J. Münch, P. Abrahamsson (Eds.), Product-Focused Software Process Improvement. XII, 414 pages. 2007.

Vol. 4574: J. Derrick, J. Vain (Eds.), Formal Techniques for Networked and Distributed Systems – FORTE 2007. XI, 375 pages. 2007.

Vol. 4556: C. Stephanidis (Ed.), Universal Access in Human-Computer Interaction, Part III. XXII, 1020 pages. 2007.

Vol. 4555: C. Stephanidis (Ed.), Universal Access in Human-Computer Interaction, Part II. XXII, 1066 pages. 2007.

Vol. 4554: C. Stephanidis (Ed.), Universal Acess in Human Computer Interaction, Part I. XXII, 1054 pages. 2007.

Vol. 4553: J.A. Jacko (Ed.), Human-Computer Interaction, Part IV. XXIV, 1225 pages. 2007.

Vol. 4552: J.A. Jacko (Ed.), Human-Computer Interaction, Part III. XXI, 1038 pages. 2007.

Vol. 4551: J.A. Jacko (Ed.), Human-Computer Interaction, Part II. XXIII, 1253 pages. 2007.

Vol. 4550: J.A. Jacko (Ed.), Human-Computer Interaction, Part I. XXIII, 1240 pages. 2007.

Vol. 4542: P. Sawyer, B. Paech, P. Heymans (Eds.), Requirements Engineering: Foundation for Software Quality. IX, 384 pages. 2007.

Vol. 4536: G. Concas, E. Damiani, M. Scotto, G. Succi (Eds.), Agile Processes in Software Engineering and Extreme Programming. XV, 276 pages. 2007.

Vol. 4530: D.H. Akehurst, R. Vogel, R.F. Paige (Eds.), Model Driven Architecture - Foundations and Applications. X, 219 pages. 2007.

Vol. 4523: Y.-H. Lee, H.-N. Kim, J. Kim, Y.W. Park, L.T. Yang, S.W. Kim (Eds.), Embedded Software and Systems. XIX, 829 pages. 2007.

Vol. 4498: N. Abdennahder, F. Kordon (Eds.), Reliable Software Technologies - Ada-Europe 2007. XII, 247 pages. 2007.

Vol. 4486: M. Bernardo, J. Hillston (Eds.), Formal Methods for Performance Evaluation. VII, 469 pages. 2007.

Vol. 4470: Q. Wang, D. Pfahl, D.M. Raffo (Eds.), Software Process Dynamics and Agility. XI, 346 pages. 2007.

Vol. 4468: M.M. Bonsangue, E.B. Johnsen (Eds.), Formal Methods for Open Object-Based Distributed Systems. X, 317 pages. 2007.

Vol. 4467: A.L. Murphy, J. Vitek (Eds.), Coordination Models and Languages. X, 325 pages. 2007.

Vol. 4454: Y. Gurevich, B. Meyer (Eds.), Tests and Proofs. IX, 217 pages. 2007.

Vol. 4444: T. Reps, M. Sagiv, J. Bauer (Eds.), Program Analysis and Compilation, Theory and Practice. X, 361 pages. 2007.

Vol. 4440: B. Liblit, Cooperative Bug Isolation. XV, 101 pages. 2007.

Vol. 4408: R. Choren, A. Garcia, H. Giese, H.-f. Leung, C. Lucena, A. Romanovsky (Eds.), Software Engineering for Multi-Agent Systems V. XII, 233 pages. 2007.

Vol. 4406: W. De Meuter (Ed.), Advances in Smalltalk. VII, 157 pages. 2007.

Vol. 4405: L. Padgham, F. Zambonelli (Eds.), Agent-Oriented Software Engineering VII. XII, 225 pages. 2007.

Vol. 4401: N. Guelfi, D. Buchs (Eds.), Rapid Integration of Software Engineering Techniques. IX, 177 pages. 2007.

Vol. 4385: K. Coninx, K. Luyten, K.A. Schneider (Eds.), Task Models and Diagrams for Users Interface Design. XI, 355 pages. 2007.

Vol. 4383: E. Bin, A. Ziv, S. Ur (Eds.), Hardware and Software, Verification and Testing. XII, 235 pages. 2007.

Vol. 4379: M. Südholt, C. Consel (Eds.), Object-Oriented Technology. VIII, 157 pages. 2007.

Vol. 4364: T. Kühne (Ed.), Models in Software Engineering. XI, 332 pages. 2007.

Vol. 4355: J. Julliand, O. Kouchnarenko (Eds.), B 2007: Formal Specification and Development in B. XIII, 293 pages. 2006.

Vol. 4354: M. Hanus (Ed.), Practical Aspects of Declarative Languages. X, 335 pages. 2006.

Vol. 4350: M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, C. Talcott, All About Maude - A High-Performance Logical Framework. XXII, 797 pages. 2007.

Vol. 4348: S. Tucker Taft, R.A. Duff, R.L. Brukardt, E. Plödereder, P. Leroy, Ada 2005 Reference Manual. XXII, 765 pages. 2006.

Vol. 4346: L. Brim, B.R. Haverkort, M. Leucker, J. van de Pol (Eds.), Formal Methods: Applications and Technology. X, 363 pages. 2007.

Vol. 4344: V. Gruhn, F. Oquendo (Eds.), Software Architecture. X, 245 pages. 2006.

Vol. 4340: R. Prodan, T. Fahringer, Grid Computing. XXIII, 317 pages. 2007.

Vol. 4336: V.R. Basili, H.D. Rombach, K. Schneider, B. Kitchenham, D. Pfahl, R.W. Selby (Eds.), Empirical Software Engineering Issues. XVII, 193 pages. 2007.

Vol. 4326: S. Göbel, R. Malkewitz, I. Iurgel (Eds.), Technologies for Interactive Digital Storytelling and Entertainment. X, 384 pages. 2006.

Vol. 4323: G. Doherty, A. Blandford (Eds.), Interactive Systems. XI, 269 pages. 2007.

Vol. 4322: F. Kordon, J. Sztipanovits (Eds.), Reliable Systems on Unreliable Networked Platforms. XIV, 317 pages. 2007.

Vol. 4309: P. Inverardi, M. Jazayeri (Eds.), Software Engineering Education in the Modern Age. VIII, 207 pages. 2006.

Vol. 4294: A. Dan, W. Lamersdorf (Eds.), Service-Oriented Computing – ICSOC 2006. XIX, 653 pages. 2006.

Vol. 4290: M. van Steen, M. Henning (Eds.), Middleware 2006. XIII, 425 pages. 2006.

Vol. 4279: N. Kobayashi (Ed.), Programming Languages and Systems. XI, 423 pages. 2006.

Vol. 4262: K. Havelund, M. Núñez, G. Roşu, B. Wolff (Eds.), Formal Approaches to Software Testing and Runtime Verification. VIII, 255 pages. 2006.

Vol. 4260: Z. Liu, J. He (Eds.), Formal Methods and Software Engineering. XII, 778 pages. 2006.

¥565.00元

Table of Contents

Testing

Program Verification

Trust and Security

Models of Computation

Distributed Programming

Model-Based Testing of Environmental Conformance of Components

Lars Frantzen[1,2] and Jan Tretmans[2,3]

[1] Instituto di Scienza e Tecnologie della Informazione "Alessandro Faedo"
Consiglio Nazionale delle Ricerche, Pisa – Italy
lars.frantzen@isti.cnr.it
[2] Institute for Computing and Information Sciences
Radboud University Nijmegen – The Netherlands
{lf,tretmans}@cs.ru.nl
[3] Embedded Systems Institute
Eindhoven – The Netherlands
jan.tretmans@esi.nl

Abstract. In component-based development, the correctness of a system depends on the correctness of the individual components and on their interactions. Model-based testing is a way of checking the correctness of a component by means of executing test cases that are systematically generated from a model of the component. This model should include the behaviour of how the component can be invoked, as well as how the component itself invokes other components. In many situations, however, only a model that specifies how others can use the component, is available. In this paper we present an approach for model-based testing of components where only these available models are used. Test cases for testing whether a component correctly reacts to invocations are generated from this model, whereas the test cases for testing whether a component correctly invokes other components, are generated from the models of these other components. A formal elaboration is given in the realm of labelled transition systems. This includes an implementation relation, called **eco**, which formally defines when a component is correct with respect to the components it uses, and a sound and exhaustive test generation algorithm for **eco**.

1 Introduction

Software testing involves checking of desired properties of a software product by systematically executing the software, while stimulating it with test inputs, and observing and checking the execution results. Testing is a widely used technique to assess the quality of software, but it is also a difficult, error-prone, and labor-intensive technique. Consequently, test automation is an important area of research and development: without automation it will not be feasible to test future generations of software products in an effective and efficient manner. Automation of the testing process involves automation of the execution of test cases, automation of the analysis of test results, as well as automation of the generation of sufficiently many and valid test cases.

F.S. de Boer et al. (Eds.): FMCO 2006, LNCS 4709, pp. 1–25, 2007.

Model-Based Testing. One of the emerging and promising techniques for test automation is model-based testing. In model based testing, a *model* of the desired behavior of the *implementation under test* (IUT) is the starting point for test generation and serves as the oracle for test result analysis. Large amounts of test cases can, in principle, be algorithmically and completely automatically generated from the model. If this model is valid, i.e., expresses precisely what the implementation under test should do, all these tests are valid, too. Model-based testing has recently gained increased attention with the popularization of modeling itself.

Most model-based testing methods deal with black-box testing of functionality. This implies that the kind of properties being tested concern the functionality of the system. Functionality properties express whether the system correctly does what it should do in terms of correct responses to given stimuli, as opposed to, e.g., performance, usability, or reliability properties. In black-box testing, the specification is the starting point for testing. The specification prescribes what the IUT should do, and what it should not do, in terms of the behavior observable at its external interfaces. The IUT is seen as a black box without internal detail, as opposed to white-box testing, where the internal structure of the IUT, i.e., the program code, is the basis for testing. Also in this paper we will restrict ourselves to black-box testing of functionality properties.

Model-based testing with labelled transition systems. One of the formal theories for model-based testing uses labelled transition systems as models, and a formal implementation relation called **ioco** for defining conformance between an IUT and a specification [10,11]. A labelled transition system is a structure with states representing the states of the system, and with transitions between states representing the actions that the system may perform. The implementation relation **ioco** expresses that an IUT conforms to its specification if the IUT never produces an output that cannot be produced by the specification. In this theory, an algorithm for the generation of test cases exists, which is provably sound for **ioco**-conformance, i.e., generated test cases only detect **ioco** errors, and exhaustive, i.e., all potential **ioco** errors can be detected.

Testing of Components. In component-based development, systems are built by gluing components together. Components are developed separately, often by different manufacturers, and they can be reused in different environments. A component is responsible for performing a specific task, or for delivering a specified service. A user requesting this service will invoke the component to provide its service. In doing so, the component may, in turn, invoke other components for providing their services, and these invoked components may again use other components. A component may at the same time act as a service provider and as a service requester.

A developer who composes a system from separate components, will only know about the services that the components perform, and not about their internal details. Consequently, clear and well-specified interfaces play a crucial role in

component technology, and components shall correctly implement these interface specifications. Correctness involves both the component's role as a service provider and its role as a service requester: a component must correctly provide its specified service, as well as correctly use other components.

Component-based testing. In our black-box setting, component-based testing concerns testing of behavior as it is observed at the component's interfaces. This applies to testing of individual components as well as to testing of aggregate systems built from components, and it applies to testing of provided services, as well as to testing of how other services are invoked.

When testing aggregated systems this can be done "bottom-up", i.e., starting with testing the components that do not invoke other components, and then adding components to the system that use the components already tested, and so forth, until the highest level has been reached. Another approach is to use *stubs* to simulate components that are invoked, so that a component can be tested without having the components available that are invoked by the component under test.

Model-based testing of components. For model-based testing of an individual component, we, in principle, need a complete model of the component. Such a model should specify the behavior at the service providing interface, the behavior at the service requesting interface, and the mutual dependencies between actions at both interfaces. Such a complete model, however, is often not available. Specifications of components are usually restricted to the behavior of the provided services. The specification of how other components are invoked is considered an internal implementation detail, and, from the point of view of a user of an aggregate system, it is.

Goal. The aim of this paper is to present an approach for model-based testing of a component at both the service providing interface and the requesting interface in a situation where a complete behavior model is not available. The approach assumes that a specification of the provided service is available for both the component under test, and for the components being invoked by the component under test. Test cases for the provided service are derived from the corresponding service specification. Test cases for checking how the component requests services from other components are derived from the provided service specifications of these other components.

The paper builds on the **ioco**-test theory for labelled transition systems, it discusses where this theory is applicable for testing components, and where it is not. A new implementation relation is introduced called *environmental conformance* – **eco**. This relation expresses that a component correctly invokes another component according to the provided service specification of that other component. A complete (sound and exhaustive) test generation algorithm for **eco** is given.

Overview. Section 2 starts with recalling the most important concepts of the **ioco**-test theory for labelled transition systems, after which Section 3 sets the

scene for formally testing components. The implementation relation **eco** is introduced in Section 4, followed by the test generation algorithm in Section 5. The combination of testing at different interfaces is briefly discussed in Section 6. Concluding remarks are presented in Section 7.

2 Testing for Labelled Transition Systems

Model-based testing deals with models, correctness (or conformance-) relations, test cases, test generation algorithms, and soundness and exhaustiveness of the generated test cases with respect to the conformance relations. This section presents the formal test theory for labelled transition systems using the **ioco**-conformance relation; see [10,11]. This theory will be our starting point for the discussion of model-based testing of components in the next sections.

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

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

The labels in L_I and L_U represent the inputs and outputs, respectively, of a system, i.e., the system's possible interactions with its environment[1]. Inputs are usually decorated with '?' and outputs with '!'. We use $L = L_I \cup L_U$ when we abstract from the distinction between inputs and outputs.

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

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

[1] The 'U' refers to 'uitvoer', the Dutch word for 'output', which is preferred for historical reasons, and to avoid confusion between L_O (letter 'O') and L_0 (digit zero).

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

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

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

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

1. $traces(p) =_{\text{def}} \{ \sigma \in L^* \mid p \stackrel{\sigma}{\Longrightarrow} \}$
2. p **after** $\sigma =_{\text{def}} \{ p' \mid p \stackrel{\sigma}{\Longrightarrow} p' \}$
3. P **after** $\sigma =_{\text{def}} \bigcup \{ p \textbf{ after } \sigma \mid p \in P \}$
4. P **refuses** $A =_{\text{def}} \exists p \in P, \; \forall \mu \in A \cup \{\tau\} : \; p \xrightarrow{\mu} \!\!\!\!/$

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

Representing labelled transition systems. To represent labelled transition systems we use either graphs (as in Fig. 1), or expressions in a process-algebraic-like language with the following syntax:

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

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

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

Table 1. Structural operational semantics

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

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

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

Input-output transition systems. In model-based testing there is a specification, which prescribes what an IUT shall do, and there is the IUT itself which is a black-box performing some behavior. In order to formally reason about the IUT's behavior the assumption is made that the IUT behaves as if it were some kind of formal model. This assumption is sometimes referred to as the test assumption or test hypothesis.

In the **ioco**-test theory a specification is a labelled transition system in $\mathcal{LTS}(L_I, L_U)$. An implementation is assumed to behave as if it were a labelled transition system that is always able to perform any input action, i.e., all inputs are enabled in all states. Such a system is defined as an *input-output transition system.* The class of such input-output transition systems is denoted by $\mathcal{IOTS}(L_I, L_U) \subseteq \mathcal{LTS}(L_I, L_U)$.

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

$$\forall \sigma, q:\ q_0 \stackrel{\sigma}{\Longrightarrow} q \ \text{ implies } \ \forall a{\in}L_I:\ q \stackrel{a}{\Longrightarrow}$$

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

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

1. *A state* q *of* p *is* quiescent, *denoted by* $\delta(q)$, *if* $\forall \mu{\in}L_U \cup \{\tau\}:\ q \not\xrightarrow{\mu}$

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 2 and 3 also apply to transition systems with label set L_δ.

The implementation relation **ioco**. An implementation relation is intended to precisely define when an implementation is correct with respect to a specification. The first implementation relation that we consider 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 (including quiescence) from i that is foreseen by s. We define **ioco** as a special case of the more general class of relations $\mathbf{ioco}_\mathcal{F}$, where $\mathcal{F} \subseteq L_\delta^*$ is a set of suspension traces, which typically depends on the specification s.

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

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

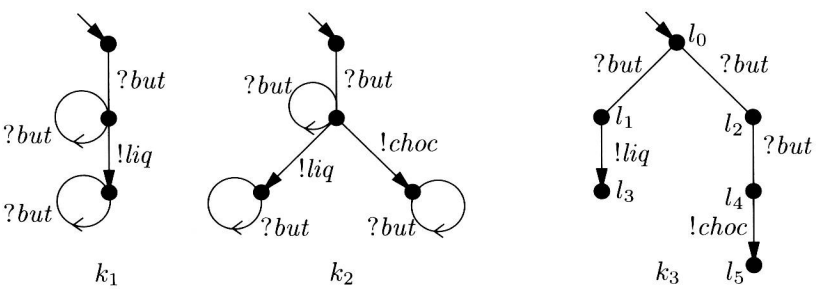

Fig. 1. Example labelled transition systems

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

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