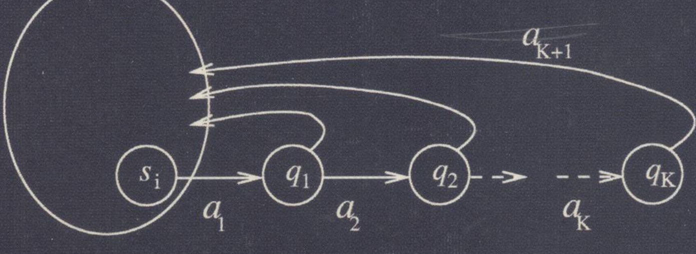
Tutorial

LNCS 3472

Manfred Broy
Bengt Jonsson
Joost-Pieter Katoen
Martin Leucker
Alexander Pretschner (Eds.)

Model-Based Testing of Reactive Systems

Advanced Lectures



Springer

TP306
M689

Manfred Broy Bengt Jonsson
Joost-Pieter Katoen Martin Leucker
Alexander Pretschner (Eds.)

Model-Based Testing of Reactive Systems

Advanced Lectures

Springer

Volume Editors

Manfred Broy
Martin Leucker
TU Munich
Institute for Informatics I4
Boltzmannstr. 3, 85748 Garching, Germany
E-mail: {broy,leucker}@in.tum.de

Bengt Jonsson
Uppsala University
Department of Computer Systems
Box 337, 751 05 Uppsala, Sweden
E-mail: bengt@it.uu.se

Joost-Pieter Katoen
University of Twente
Department of Computer Science
P.O. Box 271, 7500 AE Enschede, The Netherlands
E-mail: katoen@cs.utwente.nl

Alexander Pretschner
ETH Zurich
D-INFK, Information Security
Haldeneggsteig 4, 8092 Zürich, Switzerland
E-mail: alexander.pretschner@inf.ethz.ch

Library of Congress Control Number: 2005927641

CR Subject Classification (1998): D.2.5, D.2.4, D.2, F.3.1, D.2.11, D.3.1

ISSN 0302-9743
ISBN-10 3-540-26278-4 Springer Berlin Heidelberg New York
ISBN-13 978-3-540-26278-7 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Boller Mediendesign
Printed on acid-free paper SPIN: 11498490 06/3142 5 4 3 2 1 0

¥651.36元

Preface

Testing is the primary hardware and software verification technique used by industry today. Usually, it is ad hoc, error prone, and very expensive. In recent years, however, many attempts have been made to develop more sophisticated, formal testing methods. But a comprehensive account of the area of formal testing is missing. The goal of this seminar volume is to provide an in-depth exposure of this emerging area, especially to make it easily accessible to new researchers in this field.

Since testing describes the methodology used to obtain better systems, it is widely used in many scientific and industrial sectors dealing with system development. As such, a book on testing is hardly ever a comprehensive overview on the whole domain of testing, but a selection of important approaches and application domains. In this volume, we focus on testing methods for *reactive systems*. By reactive systems, we understand software and hardware systems with a (usually) non-terminating behavior that interact through visible events, such as Web servers, communication protocols, operating systems, smart cards, processors, etc.

Furthermore, in most chapters of this book, we follow the so-called *model-based* approach. The underlying assumption in the model-based approach is the existence of a precise formal model of the system being developed. This model can be used for studying the system to be. Especially in the testing phase of product development, it can be used to generate *complete test suites* to show *conformance* of the model and the actual implementation, or, just to derive "interesting" *test cases* to check the developed system.

The 19 chapters of the book are grouped into six parts. In the first part, we present the approaches for *testing for finite-state machines*, also called *Mealy machines*. The second part, called *testing of labeled transition systems*, gives an overview of the testing theory due to Hennessy and De Nicola together with its extensions to I/O, timed, and probabilistic systems. In Part III, we focus on methodology, algorithms, and techniques for *model-based test case generation.*

The methods illustrated in the first three parts of the book led to the development of test tools and have been applied in many case studies showing their advantages and drawbacks. Several tools and case studies are presented in Part IV.

While test case generation can be considered the heart of testing, the testing process as a whole is more complicated. The test cases have to executed on the system under test. In several application domains, test suites are used to show conformance to a standard. For this, test cases have to be interchanged among developers. Furthermore, testing should be included in the overall development process. In Part V, called *Standardized Test Notation and Execution Architecture* we cover recent developments.

The last part of the book introduces two extensions of the typical testing approach. It describes methods for the continuous testing effort, also at a later run-time of the system. Furthermore, it recalls essentials of model checking, a different powerful technique to get "better" systems, on the one hand to separate model checking and testing, on the other hand to show possible combination leading to approaches like *black box checking* or *adaptive model checking.* We meaningfully term this last part *Beyond Testing.*

The volume is the outcome of a research seminar that was held in Schloss Dagstuhl in January 2004 and that took place as part of the so-called GI/Research Seminar series. Thirty three young researchers participated in the seminar; each of them prepared a presentation based on one or several recent articles, reshaping the material in form with special emphasis on motivation, examples, and also exercises.

Thanks are due to the International Conference and Research Center of Dagstuhl and the "Gesellschaft für Informatik (GI)" for the support it provided. Further funding was provided by the Research Training Network GAMES financed by the European Commission under the Fifth Framework Programme. We also would like to thank the authors of the tutorial papers as well as their reviewers for their efforts. Last but not least, we would like to thank Britta Liebscher and Springer for substantial help in technical and editorial matters.

The editors hope that this book will help many readers to enter the domain of model-based testing, either to apply the so-far-developed techniques to enhance their product under development, or to improve the current testing techniques to make them even more efficient and effective.

Munich, Uppsala, Enschede, Zurich, January 2005

Manfred Broy
Bengt Jonsson
Joost-Pieter Katoen
Martin Leucker
Alexander Pretschner

Contents

Part I

Testing of Finite State Machines

The first part of the book is devoted to the problem of black-box testing of finite state machines in order to discover properties of their behavior and to check that they conform to given specifications.

Finite state machines is a widely used model for reactive systems. It has been used to model systems in a wide variety of areas, including sequential circuits, communication protocol entities, and embedded controllers. The study of testing of finite state machines has been motivated as fundamental research in computer science, and by applications in the testing of sequential circuits, communication protocols, embedded controllers, etc. For the case of sequential circuits, there were activities in the 60's and 70's. Since the 80's, there has been quite a lot of activity motivated by the problem of conformance testing for communication protocols. This area has generated invaluable insights into the problem of testing the reactive aspects of systems, which can be used in testing today's complex reactive systems.

Although FSM notation is simple, conformance testing for FSMs is very useful in practice. FSMs have been widely used to directly specify many types of systems, including protocols, embedded control systems and digital circuits. Moreover, many formal notations are very similar to finite state machines, or use FSMs to specify some parts of the systems. Such notations include StateCharts [Har87], SDL for communication protocols [BH89], UML state diagrams [RJB99] and ASM [Gur94] for software, and StateFlow [Sta] and SCR [HJL96] for reactive systems. Note that many control or reactive systems are described by an infinite set of states and infinite transitions. However, it is often possible to extract for such systems a system part or a particular system behavior or interaction with the environment, which can be modeled by a finite state machine. In many case this can be done through an abstraction process that allows the designer to ignore some details and focus on the finite part of the system interaction with the environment. This finite part of the specification can be used to derive tests and to test the whole system. To apply these tests to the complete system, we have to assume that we know the input and output finite vocabulary, and that the system produces a response to an input signal within a known, finite amount of time. A state of the system can be defined as a stable condition in which it has produced the expected output and is waiting for a new input. A transition is defined as the consumption of an input, the generation of an output, and the possible move to a new state. In this chapter we consider only deterministic systems, i.e. that produce the outputs and move to the next state in a deterministic way.

Typical problems from applications are as follows.

- *Conformance testing:* Check that a finite state machine conforms to a given specification. Typically the specification is given as a finite machine, and the conformance testing is to check whether the system under test is equivalent to its specification, or that it implements it in the sense of some preorder relation.
- *Property checking:* Check that the behavior of a finite state machine satisfies certain properties. These properties can be formulated, e.g., in some temporal logic.
- *Automata Learning:* Given a finite state machine, determine its behavior. This is a harder problem, which is considered in Section 19.4 of this volume.

In this Chapter, we will focus on the problem of conformance testing. There is a wide literature on conformance testing, especially in the area of communication protocol testing. Most of these algorithms combine techniques for attacking subproblems that investigating particular states or transitions of a finite state machine. We will therefore first consider these subproblems and techniques for their solution in Chapters 1 – 3. Chapter 4 will discuss how they can be combined to the problem of testing conformance.

The contents of the respective chapters are as follows.

- Chapter 1 considers the construction of *Homing and Synchronizing Sequences:* given a finite state machine with known states and transitions, a synchronizing sequence is a sequence of input symbols which takes the machine to a unique final state, independent of the starting state. A homing sequence is a sequence such that the final state (which need not be unique) can be uniquely determined by observing the output.
- Chapter 2 considers the problem of *State Identification:* Given a finite state machine with known states and transitions, identify in which state the machine currently is.
- Chapter 3 considers the problem of *State Verification:* Given a finite state machine with known states and transitions, verify that a machine is in a particular state Björklund (36)

- Chapter 4 considers the problem of *Conformance Testing* is considered: Check that a finite state machine conforms to a given specification, given as a finite state machine.

Many works in the literature on testing of finite state machine assume that systems are be modeled as Mealy machines. Mealy machines allow to model both inputs and outputs as part of their behavior, and are therefore a suitable abstract model of communication protocol entities and other types of reactive systems. An overview of testing finite state machines is given in [LY94, LY96], from which much of the material for this section is taken. Overviews of conformance testing for communication protocols can be found in [SL89, Hol91, Lai02].

The basic concepts of finite states machines used in the following chapters is given in Appendix 21.

1 Homing and Synchronizing Sequences

Sven Sandberg

Department of Information Technology
Uppsala University
svens@it.uu.se

1.1 Introduction

1.1.1 Mealy Machines

This chapter considers two fundamental problems for Mealy machines, i.e., finite-state machines with inputs and outputs. The machines will be used in subsequent chapters as models of a system or program to test. We repeat Definition 21.1 of Chapter 21 here: readers already familiar with Mealy machines can safely skip to Section 1.1.2.

Definition 1.1. A Mealy Machine is a 5-tuple $\mathcal{M} = \langle I, O, S, \delta, \lambda \rangle$, where I, O and S are finite nonempty sets, and $\delta : S \times I \to S$ and $\lambda : S \times I \to O$ are total functions.

The interpretation of a Mealy machine is as follows. The set S consists of "states". At any point in time, the machine is in one state $s \in S$. It is possible to give inputs to the machine, by applying an input letter $a \in I$. The machine responds by giving output $\lambda(s, a)$ and transforming itself to the new state $\delta(s, a)$. We depict Mealy machines as directed edge-labeled graphs, where S is the set of vertices. The outgoing edges from a state $s \in S$ lead to $\delta(s, a)$ for all $a \in I$, and they are labeled "a/b", where a is the input symbol and b is the output symbol $\lambda(s, a)$. See Figure 1.1 for an example.

We say that Mealy machines are *completely specified*, because at every state there is a next state for every input (δ and λ are total). They are also *deterministic*, because only one next state is possible.

Applying a string $a_1 a_2 \cdots a_k \in I^*$ of input symbols starting in a state s_1 gives the sequence of states $s_1, s_2, \ldots, s_{k+1}$ with $s_{j+1} = \delta(s_j, a_j)$. We extend the transition function to $\delta(s_1, a_1 a_2 \cdots a_k) \overset{\text{def}}{=} s_{k+1}$ and the output function to $\lambda(s_1, a_1 a_2 \cdots a_k) \overset{\text{def}}{=} \lambda(s_1, a_1)\lambda(s_2, a_2) \cdots \lambda(s_k, a_k)$, i.e., the concatenation of all outputs. Moreover, if $Q \subseteq S$ is a set of states then $\delta(Q, x) \overset{\text{def}}{=} \{\delta(s, x) : s \in Q\}$. We sometimes use the shorthand $s \xrightarrow{a} t$ for $\delta(s, a) = t$, and if in addition we know that $\lambda(s, a) = b$ then we write $s \xrightarrow{a/b} t$. The number $|S|$ of states is denoted n.

Throughout this chapter we will assume that an explicit Mealy machine $\mathcal{M} = \langle I, O, S, \delta, \lambda \rangle$ is given.

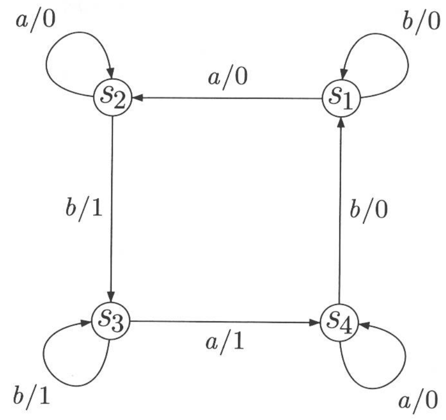


Fig. 1.1. A Mealy machine $\mathcal{M} = \langle I, O, S, \delta, \lambda \rangle$ with states $S = \{s_1, s_2, s_3, s_4\}$, input alphabet $I = \{a, b\}$, and output alphabet $O = \{0, 1\}$. For instance, applying a starting in s produces output $\lambda(s, a) = 0$ and moves to next state $\delta(s, a) = t$.

1.1.2 Synchronizing Sequences

In the problems of this chapter, we do not know the initial state of a Mealy machine and want to apply a sequence of input symbols so that the *final* state becomes known. A **synchronizing sequence** is one that takes the machine to a unique final state, and this state does not depend on where we started. Which particular final state is not specified: it is up to whoever solves the problem to select it. Thus, formally we have:

Definition 1.2. A sequence $x \in I^*$ is **synchronizing** (for a given Mealy machine) if $|\delta(S, x)| = 1$. [1] □

Note that synchronizing sequences are independent of the output. Consequently, when talking about synchronizing sequences we will sometimes omit stating the output of the machine. For the same reason, it is not meaningful to talk about synchronizing sequences "for minimized machines", because if we ignore the output then all machines are equivalent.

Example 1.3. Synchronizing sequences have many surprising and beautiful applications. For instance, robots that grasp and pick up objects, say, in a factory, are often sensitive to the orientation of the object. If objects are fed in a random

[1] The literature uses an amazing amount of synonyms (none of which we will use here), including *synchronizing word* [KRS87], *synchronization sequence* [PJH92], *reset sequence* [Epp90], *reset word* [Rys97], *directing word* [ČPR71], *recurrent word* [Rys92], and *initializing word* [Göh98]. Some texts talk about the machine as being a *synchronized* [CKK02], *synchronizing* [KRS87], *synchronizable* [PS01], *resettable* [PJH92], *reset* [Rys97], *directable* [BIĆP99], *recurrent* [Rys92], *initializable* [Göh98], *cofinal* [ID84] or *collapsible* [Fri90] *automaton.*

orientation, the problem arises of how to rotate them from an initially unknown orientation to a known one. Using sensors for this is expensive and complicated. A simple and elegant solution is depicted in Figure 1.2. Two parallel "pushing walls" are placed around the object, and one is moved toward the other so that it starts pushing the object, rotating it until a stable position between the walls is reached. Given the possible directions of these pushing walls, one has to find a sequence of pushes from different directions that takes the object to a known state. This problem can be reduced to finding a synchronizing sequence in a machine where the states are the possible orientations, the input alphabet is the set of possible directions of the walls, and the transition function is given by how a particular way of pushing rotates the object into a new orientation. This problem has been considered by, e.g., Natarajan [Nat86] and Eppstein [Epp90], who relate the problem to automata but use a slightly different way of pushing. Rao and Goldberg [RG95] use our way of pushing and their method works for more generally shaped objects. □

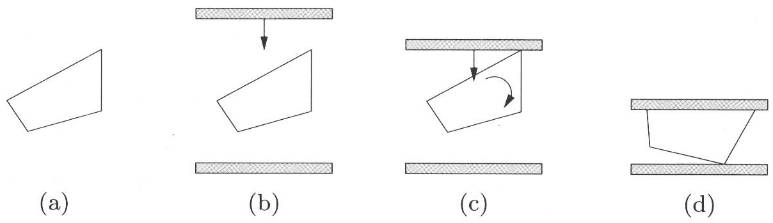


Fig. 1.2. Two pushing walls rotating the object to a new position. (a) The object. (b) One wall moves toward the object until (c) it hits it and starts pushing it, rotating it to the final stable position (d).

An alternative way to formulate the synchronizing sequence problem is as follows. Let S be a finite set, and $f_1, \ldots, f_k : S \to S$ total functions. Find a composition of the functions that is constant. Function f_i corresponds to $\delta(\cdot, a_i)$, where a_i is the i'th input symbol.

Example 1.4. To see that synchronizing sequences do not always exist, consider the Mealy machine in Figure 1.1. If the same sequence of input symbols is applied to two states that are "opposite corners", then the respective final states will be opposite corners too. So in particular no sequence x satisfies $\delta(s_1, x) = \delta(s_3, x)$ or $\delta(s_2, x) = \delta(s_4, x)$. □

Besides the parts orienting problem in Example 1.3, synchronizing sequences have been used to generate test cases for synchronous circuits with no reset [CJSP93], and are also important in theoretical automata theory and structural theory of many-valued functions [Sal02].