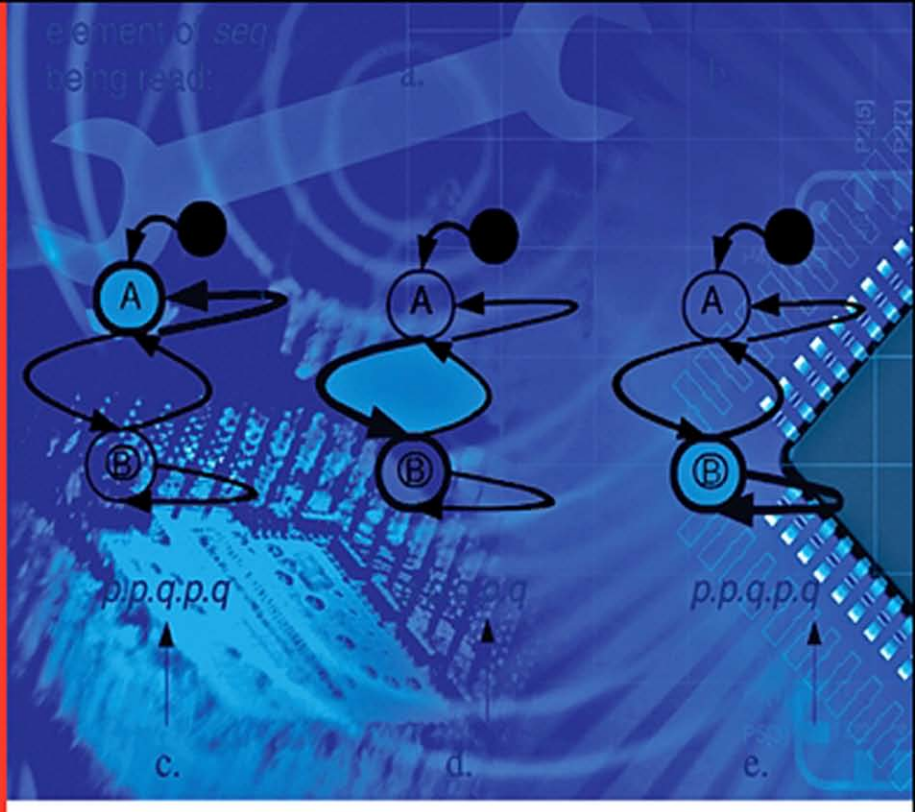

Modeling and Verification Using UML Statecharts

A Working Guide to Reactive System Design, Runtime Monitoring and Execution-based Model Checking

CD-ROM INCLUDED

Contains

- Source Code
- Statechart Templates

Doron Drusinsky

Newnes

Modeling and Verification Using UML Statecharts

This page intentionally left blank

Modeling and Verification Using UML Statecharts

A Working Guide to Reactive System Design, Runtime Monitoring and Execution-Based Model Checking

Doron Drusinsky

ELSEVIER

AMSTERDAM • BOSTON • HEIDELBERG • LONDON
NEW YORK • OXFORD • PARIS • SAN DIEGO
SAN FRANCISCO • SINGAPORE • SYDNEY • TOKYO

Newnes is an imprint of Elsevier

Newnes is an imprint of Elsevier
30 Corporate Drive, Suite 400, Burlington, MA 01803, USA
Linacre House, Jordan Hill, Oxford OX2 8DP, UK

Library of Congress Cataloging-in-Publication Data

Drusinsky, Doron.
Modeling and verification using UML statecharts : a working guide to reactive system design, runtime monitoring, and execution-based model checking / Doron Drusinsky.
p. cm.
ISBN 0-7506-7949-2 (pbk. : alk. paper) 1. UML (Computer science) 2. Formal methods (Computer science) 3. Computer software--Development. I. Title.
QA76.76.D47D78 2006
005.1'17--dc22
2006005265

British Library Cataloguing-in-Publication Data

A catalogue record for this book is available from the British Library.
ISBN-13: 978-0-7506-7949-7
ISBN 0-7506-7949-2

For information on all Newnes publications,
visit our website at www.books.elsevier.com.

06 07 08 09 10 10 9 8 7 6 5 4 3 2 1
Printed in the United States of America.

Dedication

This book could not have been written without the lifelong support of my parents Harry and Luba.

To my wonderful children, Dana, Gabi, Shiron, and Maya with whom I learned the real meaning of reactive systems, and to my beloved wife Dganit.

This page intentionally left blank

Contents

Preface

In my twenty years of practice in the software and computer science field, I was fortunate enough to have two careers: one in the industry and one in the academia. Wearing these two hats I witnessed successful transitions of research to commercial applications, such as in the cases of cryptology and digital signal processing (DSP).

Formal methods, an assortment of mathematical methods for the specification, development, and verification of software, did not enjoy such a success. After being researched for a quarter of a century or more by some of the most brilliant minds in the world, formal methods have been adopted in a very limited manner by the industry.

From an academic perspective, the most common explanation for this lackluster acceptance is that the *problem is hard.* In other words, the problem that academic research usually tries to address—mathematically proving that a program conforms to a formal specification—is a hard problem to solve using computer-aided tools due to computer science complexity-theory related issues. I refer to this problem as the *verification problem*.

From an industry perspective, however, the core issues seem rather different. Engineers and programmers want techniques that reduce their pain or win them a gold mine, and hopefully both. It is therefore hard to sell to engineers and programmers the idea that some unknown academic—albeit mathematical—formal specification language is actually better in capturing requirements than simply coding in them directly in Java. The idea of having yet another

language one needs to master, resulting in three separate views of the component that need to be maintained and synchronized (formal specification, source code, and UML) is hard to sell without having a clear benefit as an end goal—a benefit that's hard to justify given the verification problem discussed above. In short, the prime issue seen from the industry is about specification. I call this problem the *specification problem*.

To be truthful, some formal methods have been recently accepted by the software industry; specifically, these are methods that relate to the specification and verification of *transformational components* (the distinction between transformational and reactive systems is described in Chapter 2). Techniques such as design-by-contract, manifested by the Java Modeling Language (JML), are now used by many Java developers. In fact, you may think of this book as suggesting corresponding techniques and tools suitable for *reactive components*.

This book addresses the specification problem first and foremost. The book describes UML statecharts, the primary UML language when it comes to reactive components. It then describes how to use the same diagrammatic language for specifying *requirements* for reactive components (which we call *temporal* requirements) instead of using a special academic language. Having both the component design and its formal requirement specification done in the same language highlights the primary question engineers have always asked about formal methods approach, namely: *why bother*? Why not just have one kind of statechart—the design state-chart? This is an excellent question and I devote an entire chapter to it.

The book also addresses the verification problem using run-time monitoring, a lightweight method that is admittedly not perfect but it works and scales for real systems. I then show how to extend run-time monitoring with automatic test generation for the purpose of constructing an execution-based model checker.

Acknowledgments

I am indebted, far and foremost, to Dr. D. Caffal of the MDA for his vision and courage without which many of the ideas presented in this book would not have come to light.

My colleagues at NPS, specifically Bret Michael, Man-tak shing and Tom Cook have provided me with excellent feedback and support throughout the last couple of years. Members of the MDNT, including Scott Pringle, Nick Sklavounos, Dion Hinchcliffe, Chris Kauffman, Erik Stein, Steve Apsel, Dirk Penberthy, Thad Goodwyn, and Craig Trader have all provided me with excellent feedback over the course of the last two years. Mike Robison provided excellent writing and editing support. Last but not least Dganit Drusinsky assisted me to bring this material to the light of day.

What's on the CD-ROM?

CD-ROM contains:

CD-ROM contains diagrams and code for the example in Chapter 4, and also includes runtime code for monitoring and automatic white box test generation.

Chapter 1

Formal Requirements and Finite Automata Overview

1.1. Terms

Many students consider the theory of finite automata and formal languages theoretical and irrelevant to their future livelihood. Indeed, the theory is more often than not taught as a prelude to complexity theory.

In this book we will put a fresh spin on the theory, using it as a prelude to UML-based modeling, specification, and verification of reactive systems. To be relevant to reactive systems in general, and to UML in particular, we will use domain-specific terms, listed in Table 1.1, that are not usually associated with formal languages.

TABLE 1.1 UML vs. Formal-Language terms.

Terms Used in This Book	Classical, Formal-Language Counterparts
Domain of Discourse	Alphabet(s)
Event or Condition	Alphabet letter
Scenario	String
Specification or Requirement	Formal Language

Throughout this chapter we will consider using automata and formal languages in the context of the specification or design of a software component, which we will call the *component under design*.

1.2. Finite Automata: The Basics

1.2.1. The Domain of Discourse (Alphabet)

Formally speaking, an *alphabet*, typically represented with the Greek letter $\sum$, is a finite set of symbols called *letters*. In practice, these symbols are the names of events or conditions in the domain of discourse for the component under design. For the sake of simplicity, and to be able to tie our discussion closely to the theory of formal languages, in this chapter we will mostly interpret alphabet letters as events. We will leave the distinction between conditions and events to Chapter 2, where we will see how statecharts accommodate both.

A question often asked is, *Which events do we include in the alphabet?* The answer is simple: *every* event we might need for modeling or specification. In other words, the domain of discourse, as its name suggests, contains all the events that need to be taken into account during those design phases. It is therefore important to nail down the domain of discourse before proceeding to the modeling or specification phases. All subsequent modeling and specification will be based on the domain of discourse.

Consider, for example, a traffic-light controller that receives the following inputs from its environment: *oneMinuteElapsed*, *newCar*, and *newAmbulance*. The alphabet for the controller is then $\sum_{tn1}$ = {*oneMinuteElapsed*, *newCar*, *newAmbulance*}.

For alphabets that consist of conditions, there are two approaches to the relationships among member conditions.

The first, taken by formal language theory, says that conditions (letters) in a single alphabet are by definition always mutually exclusive. More precisely, exactly one condition from the alphabet must be true at any given time. If two conditions could be true simultaneously, they must be associated with distinct alphabets. Hence, a system with three unrelated conditions C_1, C_2, and C_3 has a domain of discourse that consists of the three alphabets $\Sigma_1 = \{C_1, !C_1\}$, $\Sigma_2 = \{C_2, !C_2\}$, and $\Sigma_3 = \{C_3, !C_3\}$.

The second approach, the one we will use in the context of statecharts in Chapter 2, says that all conditions are unrelated, so that any condition can be true at any given time. This amounts to the creation of a distinct alphabet, Σ_C, for every condition, C, where $\Sigma_C = \{C, !C\}$.

Events are always considered as pair-wise mutually exclusive for reasons we will discuss in Chapter 2.

1.2.2. An Input Scenario (String)

Since they are mutually exclusive, input events arrive one at a time as inputs to the component under design. Such a sequence of inputs is called a *scenario*, also known as a *string*. In other words, a scenario is a sequence of alphabet events. Consider, for example, the following alphabet with two events, $\Sigma = \{open, close\}$. The sequences *open.close.open* and *open.open.open.close.close* are two scenarios. Note how the *sequencing operator*, a period or point, "." (also known as the *concatenation* operator) is used to represent the order of events in a scenario, such as event *close* following event *open* in the scenario *open.close*.

For historical reasons theoreticians consider an input string to reside on a device called the *input tape*. In this book, however, we will not consider events to reside anywhere. In fact, we will assume that the events are lost forever after being received and processed by the component under design.

A scenario event induces a *cycle* in the component under design. Hence, a scenario of length 100 (a sequence of 100 events) induces 100 cycles in the component under design. The length of a scenario, *seq,* is denoted $|seq|$.

The empty scenario, denoted ε, is one that contains no events. It is useful mostly for mathematical purposes, such as to construct slick proofs by mathematical induction or to create recursive definitions. Obviously, since ε contains no events, $\varepsilon.x = x.\varepsilon = x$ for every scenario.

The symbol Σ^* denotes all possible finite scenarios that can be constructed from the events of the alphabet Σ. The * operator is known as the *Kleene* star operator. Note that the empty scenario is considered a member of Σ^* whereas $\Sigma^* - \{\varepsilon\}$ is denoted $\Sigma+$. Σ^* represents all possible scenarios that can be constructed with the events of the domain of discourse.

1.2.3. A Requirement (A Formal Language)

A *requirement*, also known as a *formal language*, is a *set of scenarios* constructed from events of a given domain of discourse, Σ. In other words, any subset of Σ^* constitutes a requirement. As software component developers and designers, we are not interested in just any subset of Σ^* but rather in specific ones. Intuitively, therefore, we can see that a requirement is a specification of *legal scenarios for a component under design*. We will discuss this interpretation in Chapter 4.