Model-Based Software Testing and Analysis with C#

This book teaches model-based analysis and model-based testing, important new ways to write and analyze software specifications and designs, generate test cases, and check the results of test runs. These methods increase the automation in each of these steps, making them more timely, more thorough, and more effective.

Using a familiar programming language, testers and analysts will learn to write models that describe how a program is supposed to behave. The authors work through several realistic case studies in depth and detail, using a toolkit built on the C# language and the .NET framework. Readers can also apply the methods in analyzing and testing systems in many other languages and frameworks.

Intended for professional software developers, including testers, and for university students, this book is suitable for courses on software engineering, testing, specification, or applications of formal methods.

Jonathan Jacky is a Research Scientist at the University of Washington in Seattle. He is experienced in embedded control systems, safety-critical systems, signal processing, and scientific computing. He has taught at the Evergreen State College and has been a Visiting Researcher at Microsoft Research. He is the author of The Way of Z: Practical Programming with Formal Methods.

Margus Veanes is a Researcher in the Foundations of Software Engineering (FSE) group at Microsoft Research. His research interests include model-based software development, validation, and testing.

Colin Campbell has worked on model-based testing and analysis techniques for a number of years in industry, for companies including Microsoft Research. He is a Principal of the consulting firm Modeled Computation LLC in Seattle (www.modeled-computation.com). His current interests include design analysis, the modeling of reactive and distributed systems, and the integration of components in large systems.

Wolfram Schulte is a Research Area Manager at Microsoft Research, managing the FSE group, the Programming Languages and Methods (PLM) group, and the Software Design and Implementation (SDI) group.
Model-Based Software Testing and Analysis with C#

Jonathan Jacky  
*University of Washington, Seattle*

Margus Veanes  
*Microsoft Research, Redmond, Washington*

Colin Campbell  
*Modeled Computation LLC, Seattle, Washington*

Wolfram Schulte  
*Microsoft Research, Redmond, Washington*
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preface</td>
<td>xi</td>
</tr>
<tr>
<td>Acknowledgments</td>
<td>xv</td>
</tr>
<tr>
<td><strong>I Overview</strong></td>
<td></td>
</tr>
<tr>
<td>1 Describe, Analyze, Test</td>
<td>3</td>
</tr>
<tr>
<td>1.1 Model programs</td>
<td>4</td>
</tr>
<tr>
<td>1.2 Model-based analysis</td>
<td>5</td>
</tr>
<tr>
<td>1.3 Model-based testing</td>
<td>7</td>
</tr>
<tr>
<td>1.4 Model programs in the software process</td>
<td>8</td>
</tr>
<tr>
<td>1.5 Syllabus</td>
<td>11</td>
</tr>
<tr>
<td><strong>2 Why We Need Model-Based Testing</strong></td>
<td>13</td>
</tr>
<tr>
<td>2.1 Client and server</td>
<td>13</td>
</tr>
<tr>
<td>2.2 Protocol</td>
<td>14</td>
</tr>
<tr>
<td>2.3 Sockets</td>
<td>15</td>
</tr>
<tr>
<td>2.4 Libraries</td>
<td>15</td>
</tr>
<tr>
<td>2.5 Applications</td>
<td>20</td>
</tr>
<tr>
<td>2.6 Unit testing</td>
<td>23</td>
</tr>
</tbody>
</table>
## Contents

2.7 Some simple scenarios 25
2.8 A more complex scenario 27
2.9 Failures in the field 28
2.10 Failures explained 29
2.11 Lessons learned 29
2.12 Model-based testing reveals the defect 30
2.13 Exercises 31

3 Why We Need Model-Based Analysis 32

3.1 Reactive system 32
3.2 Implementation 34
3.3 Unit testing 41
3.4 Failures in simulation 44
3.5 Design defects 46
3.6 Reviews and inspections, static analysis 47
3.7 Model-based analysis reveals the design errors 47
3.8 Exercises 52

4 Further Reading 53

II Systems with Finite Models

5 Model Programs 57

5.1 States, actions, and behavior 57
5.2 Case study: user interface 59
5.3 Preliminary analysis 61
5.4 Coding the model program 64
<table>
<thead>
<tr>
<th>Contents</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.5 Simulation</td>
<td>70</td>
</tr>
<tr>
<td>5.6 Case study: client/server</td>
<td>72</td>
</tr>
<tr>
<td>5.7 Case study: reactive program</td>
<td>82</td>
</tr>
<tr>
<td>5.8 Other languages and tools</td>
<td>92</td>
</tr>
<tr>
<td>5.9 Exercises</td>
<td>93</td>
</tr>
<tr>
<td>6 Exploring and Analyzing Finite Model Programs</td>
<td>94</td>
</tr>
<tr>
<td>6.1 Finite state machines</td>
<td>94</td>
</tr>
<tr>
<td>6.2 Exploration</td>
<td>99</td>
</tr>
<tr>
<td>6.3 Analysis</td>
<td>106</td>
</tr>
<tr>
<td>6.4 Exercise</td>
<td>114</td>
</tr>
<tr>
<td>7 Structuring Model Programs with Features and Composition</td>
<td>115</td>
</tr>
<tr>
<td>7.1 Scenario control</td>
<td>115</td>
</tr>
<tr>
<td>7.2 Features</td>
<td>117</td>
</tr>
<tr>
<td>7.3 Composition</td>
<td>121</td>
</tr>
<tr>
<td>7.4 Choosing among options for scenario control</td>
<td>129</td>
</tr>
<tr>
<td>7.5 Composition for analysis</td>
<td>131</td>
</tr>
<tr>
<td>7.6 Exercises</td>
<td>136</td>
</tr>
<tr>
<td>8 Testing Closed Systems</td>
<td>137</td>
</tr>
<tr>
<td>8.1 Offline test generation</td>
<td>137</td>
</tr>
<tr>
<td>8.2 Traces and terms</td>
<td>139</td>
</tr>
<tr>
<td>8.3 Test harness</td>
<td>142</td>
</tr>
<tr>
<td>8.4 Test execution</td>
<td>146</td>
</tr>
</tbody>
</table>
## Contents

8.5 Limitations of offline testing 147  
8.6 Exercises 148  

9 Further Reading 150  

III Systems with Complex State  

10 Modeling Systems with Structured State 155  
10.1 “Infinite” model programs 155  
10.2 Types for model programs 157  
10.3 Compound values 157  
10.4 Case study: revision control system 169  
10.5 Exercises 181  

11 Analyzing Systems with Complex State 183  
11.1 Explorable model programs 183  
11.2 Pruning techniques 186  
11.3 Sampling 190  
11.4 Exercises 190  

12 Testing Systems with Complex State 191  
12.1 On-the-fly testing 192  
12.2 Implementation, model and stepper 194  
12.3 Strategies 199  
12.4 Coverage-directed strategies 203  
12.5 Advanced on-the-fly settings 210  
12.6 Exercises 218  

13 Further Reading 219
IV Advanced Topics

14 Compositional Modeling 223
  14.1 Modeling protocol features 223
  14.2 Motivating example: a client/server protocol 224
  14.3 Properties of model program composition 241
  14.4 Modeling techniques using composition and features 245
  14.5 Exercises 246

15 Modeling Objects 247
  15.1 Instance variables as field maps 247
  15.2 Creating instances 249
  15.3 Object IDs and composition 253
  15.4 Harnessing considerations for objects 254
  15.5 Abstract values and isomorphic states 256
  15.6 Exercises 257

16 Reactive Systems 259
  16.1 Observable actions 259
  16.2 Nondeterminism 261
  16.3 Asynchronous stepping 264
  16.4 Partial explorability 265
  16.5 Adaptive on-the-fly testing 268
  16.6 Partially ordered runs 272
  16.7 Exercises 274

17 Further Reading 275
V Appendices

A Modeling Library Reference 281
  A.1 Attributes 282
  A.2 Data types 292
  A.3 Action terms 306

B Command Reference 308
  B.1 Model program viewer, mpv 308
  B.2 Offline test generator, otg 311
  B.3 Conformance tester, ct 312

C Glossary 315

Bibliography 333
Index 341
Preface

This book teaches new methods for specifying, analyzing, and testing software. They are examples of model-based analysis and model-based testing, which use a model that describes how the program is supposed to behave. The methods provide novel solutions to the problems of expressing and analyzing specifications and designs, generating test cases, and checking the results of test runs. The methods increase the automation in each of these activities, so they can be more timely, more thorough, and (we expect) more effective. The methods integrate concepts that have been investigated in academic and industrial research laboratories for many years and apply them on an industrial scale to commercial software development. Particular attention has been devoted to making these methods acceptable to working software developers. They are based on a familiar programming language, are supported by a well-engineered technology, and have a gentle learning curve.

These methods provide more test automation than do most currently popular testing tools, which only automate test execution and reporting, but still require the tester to code every test case and also to code an oracle to check the results of every test case. Moreover, our methods can sometimes achieve better coverage in less testing time than do hand-coded tests.

Testing (i.e., executing code) is not the only assurance method. Some software failures are caused by deep errors that originate in specifications or designs. Model programs can represent specifications and designs, and our methods can expose problems in them. They can help you visualize aspects of system behavior. They can perform a safety analysis that checks whether the system can reach forbidden states, and a liveness analysis that identifies dead states from which goals cannot be reached, including deadlocks (where the program seems to stop) and livelocks (where the program cycles endlessly without making progress). Analysis uses the same model programs and much of the same technology as testing.

This book is intended for professional software developers, including testers, and for university students in computer science. It can serve as a textbook or supplementary reading in undergraduate courses on software engineering, testing,
The methods are based on executable specifications that we call model programs. To use the methods taught here, you write a model program that represents the pertinent behaviors of the implementation you wish to specify, analyze, or test. You write the model program in C#, augmented by a library of data types and custom attributes. Executing the model program is a simulation of the implementation (sometimes called an animation). You can perform more thorough analyses by using a technique called exploration, which achieves the effect of many simulation runs. Exploration is similar to model checking and can check for safety, liveness, and other properties. You can visualize the results of exploration as state transition diagrams. You can use the model program to generate test cases automatically. When you run the tests, the model can serve as the oracle (standard of correctness) that automatically checks that the program under test behaved as intended. You can generate test cases in advance and then run tests later in the usual way. Alternatively, when you need long-running tests, or you must test a reactive program that responds to events in its environment, you may do on-the-fly testing, in which the test cases are generated in response to events as the test run executes. You can use model composition to build up complex model programs by combining simpler ones, or to focus exploration and testing on interesting scenarios.

In this book, we demonstrate the methods using a framework called NModel that is built on the C# language and .NET (the implementations that are modeled and tested do not have to be written in C# and do not need to run in .NET). The NModel framework includes a library for writing model programs in C#, a visualization and analysis tool mpv (Model Program Viewer), a test generation tool otg (Offline Test Generator), and a test runner tool ct (Conformance Tester). The library also exposes the functionality of mpv, otg, ct, and more, so you may write your own tools that are more closely adapted to your environment, or that provide other capabilities. To use this technology, you must write your own model program in C# that references the NModel library. Then you can use the mpv tool to visualize and analyze the behavior of your model program, in order to confirm that it behaves as you intend, and to check it for design errors. To execute tests using the test runner ct, you must write a test harness in C# that couples your implementation to the tool. You can use the test generator otg to create tests from your model program in advance, or let ct generate the test on the fly from your model program as the test run executes. If you wish, you can write a custom strategy in C# that ct uses to maximize coverage according to criteria you define.
To use the NModel library and tools, the only additional software you need is the .NET Framework Redistributable Package (and any Windows operating system capable of running it). The NModel framework, as well as .NET, are available for download at no cost.

This book is not a comprehensive survey or comparison of the model-based testing and analysis tools developed at Microsoft Research (or elsewhere). Instead, we focus on selected concepts and techniques that we believe are the most important for beginners in this field to learn, and that make a persuasive (and reasonably short) introduction. We created the NModel library and tools to support this book (and further research). We believe that the simplicity, versatility, and transparency of this technology makes it a good platform for learning the methods and experimenting with their possibilities. However, this book is also for readers who use other tools, including Spec Explorer, which is also from Microsoft Research and is also in active development. Other tools support many of the same methods we describe here, and some that we do not discuss. This book complements the other tools’ documentation by explaining the concepts and methods common to all, by providing case studies with thorough explanations, and by showing one way (of many possible ways) that a modeling and testing framework can support the techniques that we have selected to teach here.

This book is a self-contained introduction to modeling, specifications, analysis, and testing. Readers need not have any previous exposure to these topics. Readers should have some familiarity with an object-oriented programming language such as Java, C++, or C#, as could be gained in a year of introductory computer science courses. Student readers need not have taken courses on data structures and algorithms, computing theory, programming language semantics, or software engineering. This book touches on those topics, but provides self-contained explanations. It also explains the C# language features that it uses that are not found in other popular languages, such as attributes and events.

Although this book is accessible to students, it will also be informative to experienced professionals and researchers. It applies some familiar ideas in novel ways, and describes new techniques that are not yet widely used, such as on-the-fly testing and model composition.

When used with the NModel framework, C# can express the same kind of state-based models as many formal specification languages, including Alloy, ASMs, B, Promela, TLA, Unity, VDM, and Z, and also some diagramming notations, including Statecharts and the state diagrams of UML. Exploration is similar to the analysis performed by model checkers such as Spin and SMV. We have experience with several of these notations and tools, and we believe that modeling and analysis do not have to be esoteric topics. We find that expressing the models in a familiar programming language brings them within reach of most people involved in the technical aspects of software production. We also find that focusing on testing as
one of the main purposes of modeling provides motivation, direction, and a practical emphasis that developers and testers appreciate.

This book is divided into four parts. The end of each part is an exit point; a reader who stops there will have understanding and tools for modeling, analysis, and testing up to that level of complexity. Presentation is sequential through Part III, each chapter and part is a prerequisite for all the following chapters and parts. Chapters in Part IV are independent; readers can read one, some, or all in any order.

This book provides numerous practical examples, case studies, and exercises and contains an extensive bibliography, including citations to relevant research papers and reports.
Acknowledgments

Parts of this book were written at Microsoft Research. The NModel framework was designed and implemented at Microsoft Research by Colin Campbell and Margus Veanes with graph viewing functionality by Lev Nachmanson.

The ideas in this book were developed and made practical at Microsoft Research from 1999 through 2007 in the Foundations of Software Engineering group. Contributors included Mike Barnett, Nikolaj Bjorner, Colin Campbell, Wolfgang Grieskamp, Yuri Gurevich, Lev Nachmanson, Wolfram Schulte, Nikolai Tillman, Margus Veanes, as well as many interns, in particular Juhan Ernits, visitors, university collaborators, and colleagues from the Microsoft product groups. Specific contributions are cited in the “Further readings” chapters at the end of each part.

Jonathan Jacky especially thanks Colin Campbell, who introduced him to the group; Yuri Gurevich, who invited him to be a visiting researcher at Microsoft; and Wolfram Schulte, who arranged for support and resources while writing this book. Jonathan also thanks John Sidles and Joseph Garbini at the University of Washington, who granted him permission to go on leave to Microsoft Research. Jonathan thanks his wife, Noreen, for her understanding and encouragement through this project. Jonathan’s greatest thanks go to his coauthors Colin, Margus, and Wolfram, not only for these pages but also for the years of preparatory work and thought. Each made unique and absolutely essential individual contributions, without which this book would not exist.

Margus Veanes thanks the members of the Foundations of Software Engineering group, in particular Yuri Gurevich, for laying a mathematical foundation upon which much of his work has been based, and Colin Campbell, for being a great research partner. Finally, Margus thanks his wife, Katrine, and his sons, Margus and Jaan, for their love and support.

Colin Campbell would like to thank Jim Kajiya for his technical vision and steadfast support of this project over almost a decade. Colin also acknowledges a profound debt to Yuri Gurevich for teaching him how to understand discrete systems.
as evolving algebras and to Roberta Leibovitz, whose extraordinarily keen insight was welcome at all hours of the day and night.

Wolfram Schulte thanks Wolfgang Grieskamp and Nikolai Tillmann, who designed and implemented the Abstract State Machine Language and substantial parts of Spec Explorer 2004; both tools are predecessors of the work described here. He also wants to express his gratitude, to many testers, developers, and architects in Microsoft. Without their willingness to try new research ideas, their passion to push the limits of model-based testing and analysis, and their undaunted trust in his and his coauthors’ capabilities, this book would not exist – thank you.