

---

## Foreword

Anyone who has tried to write a nontrivial piece of software knows from bitter experience that the code is not likely to work quite right after the first successful compilation; nor the second or third. Sometimes, it takes a while to discover how a seemingly correct program can fail in subtle ways. The same is of course true for commercially developed software. Small flaws can hide for years and strike at the most inconvenient moment. So we learn to backup our data and cope with the apparently inevitable. There are, however, cases where we do not have the luxury of accepting products that may be subtly flawed. In some applications, software defects can lead to a loss of life or cause significant economical damage. Given that so much of our world is now controlled by software, finding ways to make this software more reliable is perhaps the most important technical challenge of our age. So, how can we accomplish this?

Most critical software applications execute in a multithreaded environment, with numerous external dependencies: they are concurrent. It is especially the concurrency aspects – the mutual dependencies – that are difficult to get right. Fortunately, today powerful tools are available to verify the logical correctness of concurrent (distributed, parallel, or multithreaded) programs. SPIN is perhaps the leading example of such a tool. Its development dates back to roughly 1980, with a first free version publicly released in 1991. It is often considered to be one of the most powerful model checkers available.

SPIN is increasingly used in the classroom to teach concurrency and model checking techniques, but, most important, it is applied in industrial practice to solve real problems in the construction of large-scale distributed software systems. The tool has been used for the verification of everything from operating systems software and communications protocols to railway signaling systems. Some of the larger applications are especially inspiring. In

the late 90s, for instance, SPIN was used to verify the control algorithms for a large new flood control system near Rotterdam in The Netherlands. It was used between 1999 and 2001 at Bell Laboratories to verify the call processing software for a new telephone switch. And, finally, SPIN is used increasingly for a thorough verification of key control algorithms for interplanetary space missions at NASA.

So far, most publications on the SPIN system have focused on its theoretical background, with less attention being paid to routine usage and application. This book offers for the first time a comprehensive introduction to SPIN from a user's perspective. It makes the capabilities of the tool accessible to a much broader audience. All key concepts are explained step-by-step, without fuss, in a clear and instructive way that can get the reader up to speed very quickly. As such, this book has no competition. This is the best introduction to the SPIN tool.

Gerard J. Holzmann  
Pasadena, California  
May 2007

---

## Preface

Surrounded as we are by software for personal computers, electronic gadgets and entertainment websites, it is easy to lose sight of the massive amount of software embedded in critical systems. I was surprised when I found out that the computerized systems in modern cars have half a million lines of code, and that electronics account for 25% of their cost and this percentage is forecast to increase.<sup>1</sup> Perhaps it is easiest to characterize a critical system as one that must be delivered without one of those infamous “end user license agreements” that disavows liability and requires you to renounce any claim to a guarantee.

Formal methods are powerful tools in the arsenal of software engineers who develop software that *must* work correctly. While the principles of formal methods were developed decades ago by pioneers of computer science like C.A.R. Hoare and the late E.W. Dijkstra, only recently have theoretical advances and progress in the development of software tools enabled their widespread use.

### Model checking

One of the most powerful formal methods is *model checking*. In principle model checking is trivial: simply generate all possible states of a program and check that the correctness specifications hold in each state. Furthermore, generating states and checking specifications can be done mechanically by a software tool. In practice, sophisticated algorithms based upon automata theory and logic are needed to perform model checking on nontrivial programs which have billions or trillions of states.

---

<sup>1</sup> Klaus Grimm. Software technology in an automotive company – Major challenges, *Proceedings of the 25th International Conference on Software Engineering*, 2003, 498–503.

Even the best model checkers are not “plug and play”: If a program has even one 32-bit integer variable, at each location during the execution of the program that variable can give rise to as many as  $2^{32}$  different states. As the name implies, model checkers do not check *programs*, but rather *models*, which are high level descriptions of a system. The challenge for a software engineer is to develop a model that faithfully represents the system, while at the same time remaining sufficiently concise to enable its correctness to be checked with the available resources. The complementary challenge for designers of model checkers is to include sufficiently expressive constructs to facilitate the construction of faithful models, while leaving out constructs that cannot be efficiently implemented.

## SPIN

SPIN is a model checker developed by Gerard J. Holzmann for verifying communications protocols. It has since become widely used in industries that build critical systems. In 2001, Holzmann received the ACM Software Systems Award for the development of SPIN. My interest in SPIN arose from my long-time engagement in teaching concurrent programming. Pieter Hartel convinced me to look into SPIN, and I found that SPIN is a very rare artifact: Although it is an industrial-strength tool, it can be easily used by students. The software is simple to install and to run, and models are written in PROMELA, which looks like a familiar programming language.

This led to my writing a new edition of the textbook *Principles of Concurrent and Distributed Programming* (Ben-Ari, 2006), which included material on SPIN. In addition, I built pedagogical software tools that leverage the capabilities of SPIN. I have come to believe that SPIN can be used to introduce students to important concepts in computer science, such as logic, automata, concurrency, nondeterminism, and program verification.

The only impediment I found to the wider use of SPIN in computer science education was the lack of an introductory book. *The Spin Model Checker: Primer and Reference Manual* (Holzmann, 2004) contains, in addition to elementary explanations, a wealth of material on the theory and implementation of SPIN, and on the design and verification of models for communications systems. As such, beginners might find it difficult to use as an introductory text.

*Principles of the Spin Model Checker* is intended as an introduction to SPIN for undergraduate students and for programmers without a strong background in formal methods. It presents the concepts of model checking, the constructs of PROMELA, and the capabilities of SPIN comprehensively, but in steps of gradually increasing difficulty. The book is self-contained, but

will probably be accessible only to readers with two or three years of programming experience. An elementary knowledge of logic – the propositional calculus – is also required; see my textbook *Mathematical Logic for Computer Science* (Ben-Ari, 2004) if you need help.

The book describes SPIN-based software tools that I have developed: the JSPIN development environment, SPINSPIDER for visualizing state diagrams, and VN for experiencing nondeterminism.

Of course, once you actually start to work with SPIN, you will want to consult *The Spin Model Checker* and the *man* pages.<sup>2</sup>

### Overview of the book

The book is organized into three parts. Chapters 1 through 5 introduce the main concepts that are needed to write models in PROMELA and to verify them with SPIN. Chapters 6 and 7 present structures in PROMELA that are essential for constructing models, while Chapters 8–11 include more advanced and optional material.

Models in SPIN are written in the PROMELA language; its syntax is based upon that of C, but it is sufficiently different that it seems worthwhile to give a gentle introduction to PROMELA and SPIN using *sequential* programs (Chapter 1). This is followed in Chapter 2 by an introduction to verification, again within the context of sequential programs. SPIN is primarily used for modeling and verifying concurrent systems, and this is presented in Chapter 3 on modeling multiprocess systems, in Chapter 4 on the synchronization of processes, and in Chapter 5 on linear temporal logic that is used to express correctness specifications in SPIN.

Chapter 6 is concerned with constructs for structuring data and programs, and Chapter 7 explains channels, which are used for modeling distributed systems, as well as for implementing data structures.

Chapter 8 diverges from the usual view of model checkers as tools for verifying concurrent and distributed systems. It shows how SPIN can be used to teach the important concept of nondeterminism that appears in many contexts in computer science, such as algorithms and automata. Chapter 9 presents advanced PROMELA constructs and Chapter 10 surveys advanced capabilities of SPIN for expressing correctness specifications and for optimizing verifications.

The book concludes with five cases studies in Chapter 11 designed to bring together the individual PROMELA programming structures that were

<sup>2</sup> *man* pages form the definitive documentation. They can be found in Chapters 16–19 of the *The Spin Model Checker* (Holzmann, 2004) and online at the SPIN website; they can also be downloaded with the SPIN distribution.

presented in isolation: (a) the implementation of a complex data structure; (b) further examples of nondeterministic algorithms; (c) a real-time scheduling algorithm; (d) a model that uses discrete time; (e) an advanced algorithm for a distributed system.

Appendix A gives an overview of the software tools I have developed. For details see the documentation included within the archive for each tool. Appendix B contains the addresses of relevant websites. A short list of references will direct you to more advanced books.

Instructions for running SPIN are given in two forms: (a) using the JSPIN environment, and (b) commands and arguments for running SPIN directly from the command line.

The source code of all the PROMELA programs in the book is available on the companion website at [www.springer.com/978-1-84628-769-5](http://www.springer.com/978-1-84628-769-5).

### Conventions

Starred (\*) sections can be skipped on your first reading and returned to later on. Framed text is used to emphasize important warnings. Passages marked **Advanced** can be safely passed over by most readers. Acronyms are used for several books referred to frequently: *SMC* for Holzmann (2004), *MLCS* for Ben-Ari (2004), and *PCDP* for Ben-Ari (2006).

### Acknowledgements

I am deeply indebted to Gerard J. Holzmann for his support and help during the writing of *PCDP* and of this book, and during the development of my software tools. I am grateful to Angelika Mader for carefully reading the manuscript; her eagle eye for obscure explanations significantly improved the presentation. Thanks also to Pieter Hartel for introducing me to SPIN, Michal Armoni for collaborating on the VN software, and Dragan Bošnački for help with modeling discrete time. Finally, it has been a pleasure to work with Beverley Ford and the entire staff at Springer.

Mordechai Ben-Ari  
Rehovot  
June 2007

---

## Contents

<b>Foreword</b> .....	v
<b>Preface</b> .....	vii
<b>1 Sequential Programming in PROMELA</b> .....	1
1.1 A first program in PROMELA .....	1
1.2 Random simulation .....	2
1.3 Data types .....	4
1.3.1 Type conversions .....	6
1.4 Operators and expressions .....	6
1.4.1 Local variables .....	7
1.4.2 Symbolic names* .....	8
1.5 Control statements .....	10
1.6 Selection statements .....	10
1.6.1 Conditional expressions* .....	14
1.7 Repetitive statements .....	15
1.7.1 Counting loops .....	16
1.8 Jump statements* .....	17
<b>2 Verification of Sequential Programs</b> .....	19
2.1 Assertions .....	19
2.2 Verifying a program in SPIN .....	23
2.2.1 Guided simulation .....	26
2.2.2 Displaying a computation .....	26
<b>3 Concurrency</b> .....	29
3.1 Interleaving .....	29
3.1.1 Displaying a computation .....	31
3.2 Atomicity .....	33

3.3	Interactive simulation	34
3.4	Interference between processes	35
3.5	Sets of processes	37
3.6	Interference revisited	38
3.7	Deterministic sequences of statements*	40
3.8	Verification with assertions	42
3.9	The critical section problem	44
<b>4</b>	<b>Synchronization</b>	<b>47</b>
4.1	Synchronization by blocking	47
4.2	Executability of statements	50
4.3	State transition diagrams	51
4.4	Atomic sequences of statements	54
4.4.1	<b>d_step</b> and <b>atomic</b> *	56
4.5	Semaphores	58
4.6	Nondeterminism in models of concurrent systems	60
4.6.1	Generating values nondeterministically	61
4.6.2	Generating from an arbitrary range*	62
4.7	Termination of processes	63
4.7.1	Deadlock	63
4.7.2	End states*	64
4.7.3	The order of process termination*	66
<b>5</b>	<b>Verification with Temporal Logic</b>	<b>69</b>
5.1	Beyond assertions	69
5.2	Introduction to linear temporal logic	71
5.2.1	The syntax of LTL	71
5.2.2	The semantics of LTL	72
5.3	Safety properties	75
5.3.1	Expressing safety properties in LTL	75
5.3.2	Expressing safety properties in PROMELA	76
5.3.3	Verifying safety properties in SPIN	78
5.4	Liveness properties	80
5.4.1	Expressing liveness properties in SPIN	81
5.4.2	Verifying liveness properties in SPIN	82
5.5	Fairness	84
5.6	Duality	85
5.7	Verifying correctness without ghost variables*	86
5.8	Modeling a noncritical section*	87
5.9	Advanced temporal specifications*	89
5.9.1	Latching	89

5.9.2	Infinitely often .....	90
5.9.3	Precedence .....	91
5.9.4	Overtaking .....	92
5.9.5	Next .....	94
<b>6</b>	<b>Data and Program Structures</b> .....	<b>95</b>
6.1	Arrays .....	95
6.2	Type definitions .....	97
6.3	The preprocessor .....	99
6.3.1	Condition compilation* .....	100
6.3.2	Macros* .....	100
6.4	Inline .....	101
<b>7</b>	<b>Channels</b> .....	<b>105</b>
7.1	Channels in PROMELA .....	106
7.1.1	Channels and channel variables .....	108
7.2	Rendezvous channels .....	109
7.2.1	Reply channels .....	110
7.2.2	Arrays of channels .....	113
7.2.3	Local channels .....	114
7.2.4	Limitations of rendezvous channels .....	114
7.3	Buffered channels .....	115
7.4	Checking the content of a channel .....	116
7.4.1	Checking if a channel is full or empty .....	117
7.4.2	Checking the number of messages in a channel .....	117
7.5	Random receive* .....	119
7.6	Sorted send* .....	121
7.7	Copying the value of a message* .....	122
7.8	Polling* .....	122
7.9	Comparing rendezvous and buffered channels .....	123
<b>8</b>	<b>Nondeterminism*</b> .....	<b>125</b>
8.1	Nondeterministic finite automata .....	125
8.1.1	<b>timeout</b> .....	128
8.1.2	Using verification to find accepting computations .....	128
8.1.3	Finding all counterexamples .....	129
8.1.4	$\lambda$ -transitions .....	130
8.2	VN: Visualizing nondeterminism .....	131
8.3	$\mathcal{NP}$ problems .....	133

<b>9</b>	<b>Advanced Topics in PROMELA*</b>	137
9.1	Specifiers for variables	137
9.2	Predefined variables	138
9.2.1	The anonymous variable	138
9.2.2	Process identifiers	138
9.3	Priority	140
9.3.1	Simulation priority	140
9.3.2	Modeling priority with global constraints	140
9.4	Modeling exceptions	142
9.5	Reading from standard input	143
9.6	Embedded C code	143
<b>10</b>	<b>Advanced Topics in SPIN*</b>	145
10.1	How SPIN searches the state space	145
10.2	Optimizing the performance of verifications	148
10.2.1	Writing efficient models	148
10.2.2	Allocating memory for the hash table	149
10.2.3	Compressing the state vector	152
10.2.4	Minimal automaton	153
10.2.5	Partial-order reduction	153
10.3	Never claims	153
10.3.1	A never claim for a safety property	155
10.3.2	A never claim for a liveness property	156
10.3.3	Never claims for other LTL formulas	157
10.3.4	Predefined constructs for use in never claims	159
10.4	Non-progress cycles	159
<b>11</b>	<b>Case Studies*</b>	163
11.1	Channels as data structures	163
11.2	Nondeterministic algorithms	168
11.2.1	The eight-queens problem	168
11.2.2	Cycles in a directed graph	171
11.3	Modeling a real-time scheduling algorithm	173
11.3.1	Real-time systems	174
11.3.2	Modeling a scheduler in PROMELA	175
11.3.3	Simplifying the model	177
11.3.4	Modeling a scheduler with priorities	177
11.3.5	Rate monotonic scheduling	181
11.4	Fischer's algorithm	181
11.5	Modeling distributed systems	186
11.6	The Chandy-Lamport algorithm for global snapshots	187

11.7	The Chandy–Lamport snapshot algorithm in PROMELA . . . . .	189
11.7.1	Structure of the program . . . . .	190
11.7.2	Encoding lists of channels . . . . .	191
11.7.3	The environment node . . . . .	192
11.7.4	Local data for each node . . . . .	192
11.7.5	Nodes of the distributed system . . . . .	194
11.7.6	Nondeterministic choice of a channel . . . . .	196
11.8	Verification of the snapshot algorithm . . . . .	197
<b>A</b>	<b>Software Tools</b> . . . . .	201
A.1	SPIN . . . . .	201
A.2	JSPIN . . . . .	202
A.3	SPINSPIDER . . . . .	203
A.4	VN: Visualizing nondeterminism . . . . .	204
<b>B</b>	<b>Links</b> . . . . .	207
	<b>References</b> . . . . .	209
	<b>Index</b> . . . . .	211



