Modeling Companion for Software Practitioners

Egon Börger • Alexander Raschke

Modeling Companion for Software Practitioners

Springer

Egon Börger
Dipartimento di Informatica
Università di Pisa
Pisa, Italy

Alexander Raschke
Institute of Software Engineering
and Programming Languages
Universität Ulm
Ulm, Germany

ISBN 978-3-662-56639-8 ISBN 978-3-662-56641-1 (eBook)
https://doi.org/10.1007/978-3-662-56641-1

Library of Congress Control Number: 2018935112

Printed on acid-free paper

This Springer imprint is published by the registered company Springer-Verlag GmbH, DE part of Springer Nature.
The registered company address is: Heidelberger Platz 3, 14197 Berlin, Germany

Preface: What the Book is About

L'armonia sia non signora ma serva del oratione [1]
—The principle of *Perfettione della moderna musica*
Claudio Monteverdi

The target audiences of the book are practitioners who build software-intensive systems, but also students of the field. The book illustrates, by a variety of applications, *a modeling method which helps the practitioner to* **intellectually manage complex software-intensive systems**. This implies a four-fold support for system development, namely support to **BEDER**

Build, **E**xplain, **D**ebug (validate/verify), maintain (**E**xtend/**R**euse)

well-documented models of computational systems deemed to be reliable.

The proposed method provides this support by a combination of its abstraction concept and its operational character: models come as behavioral models in the precise and simple form of *Abstract State Machines* (ASMs), a semantically rigorously defined form of pseudo-code. In the Introduction (Chap. 1) we explain how the most general concept of abstraction inherent in ASMs permits

- to directly **adapt construction and explanation of models to** data and operations at **any desired level of abstraction**, leading from requirements to executable code,
- to rigorously **mediate between levels of abstraction**,
- to construct and debug system models **piecemeal**, in steps and componentwise,
- to establish (formulate, explain and justify) the intended **system behavior properties** at any given level of abstraction, exploiting any useful means of justification, whether by experimental **validation** or by informal, mathematical, formal or machine-supported **verification**.

[1] The music should not dictate but follow the prose.
read: The model should not dictate but reflect the problem. See [188, 189].

This book introduces into the modeling method (Part I) and into the available tool support, which makes design models machine executable and debuggable (Part II).

In Part I we show by examples how to construct, explain, debug, explore, extend and reuse accurate system design models, starting from scratch. Here we assume only elementary knowledge of common mathematical (including set-theoretic) notation, as taught in high-schools, and some basic experience with computational processes (systems, programs, algorithms). Therefore we begin with simple, rather elementary examples to introduce the **seven fundamental constructs of Abstract State Machines** which—exploiting the abstraction capabilities offered by the most comprehensive notion of ASM *state*—suffice for modeling and explaining complex sequential as well as concurrent systems:

assignment, if-then-else, parallelism, forall, choose, let, call

(Chap. 2, 3). In the remaining chapters we illustrate by some more advanced examples the wide range of applicability of the method. We include examples from automatic control systems, equipments operated by software interacting with humans, algorithms, protocols, monitoring networks of communicating agents, semantics of concurrent algorithmic languages, operating system components, software system architectures, meta-modeling for diagrammatic notations, shared memory models, distributed web services, business processes and last but not least control flow, programming and interaction patterns.[2]

Part II is written for system designers and programmers who are interested to see how the ASM modeling method can be supported by implementing **tools which make design models executable and debuggable**. This needs first of all a mathematical definition of the syntax and semantics of ASMs (Chap. 7). We then apply bootstrapping to specify in terms of basic ASMs various extensions of the seven constructs listed above, namely:

- An implementation of ASMs in CoreASM(Chap. 8)—an ASM interpreter tool which makes them machine-executable—together with an extension of CoreASM by a model debugger (Sect. 8.3).

[2] We exclude virtual machines, the semantics of common programming languages and their compilation (proven, even mechanically, to be correct!), given that there is a rich and easily available literature on challenging practical applications of the ASM method in this area. These applications include characteristic examples for the main programming paradigms: imperative, object-oriented, functional, logical etc., see the recent survey [43] for details and references. For other applications a comprehensive survey is available for the first ten years of the ASM method (up to 2003), see [76, Chap. 9]. Unfortunately the survey has not been updated so that the interested reader must search to find publications on the numerous theoretical and practical advances of the method since 2003. A good starting point are the Proceedings of the regular international ASM Workshops and since 2008 the ABZ Conferences in Springer's Lecture Notes in Computer Science [55, 239, 24, 74, 5, 66, 46, 114, 91, 205, 11, 82].

- A meta model of ASM *Control State Diagrams* (CSDs) (Chap. 9), a graphical notation which extends the well-known Finite State Machine flowchart notation to ASMs and is used throughout the book.
- Further specific modeling concepts, namely parameterized ASMs, context-aware ambient ASMs and step controlled ASMs (Chap. 4).

To illustrate how to build, debug and maintain (explore and reuse) systems and to explain their construction in a checkable manner, we construct system models from components adopting a **general, problem-oriented refinement method**. The method starts with abstract models and refines them stepwise, incrementally adding further details. This may be through adding components for additional functionality (horizontal refinements) or through specifying a component further by details leading to an implementation (vertical refinement or parameter instantiation) or any combination of the two. The chain of refinements provides a practical system decomposition and documentation which not only supports understanding, debugging (via testing or (dis-) proving behavioral properties) and maintenance (exploration or reuse of components), but it does it at any level of abstraction involved in the refinement steps. It helps to manage (understand and explain) complexity by breaking complex systems into simpler, well understood, well explained, well documented and objectively checkable parts. To achieve this, the ASM method builds upon the intimate connection between refinement and abstraction, two thought processes which complement each other.

Understanding a system is a prerequisite for justifying its well-functioning. We show in this book how the abstraction and refinement capabilities of the ASM method allow the system engineer to accompany (and document for others to check) the component building steps by **accurate intuitive explanations** of the underlying design and implementation ideas. By its abstraction/refinement pair, the ASM method provides the practitioner with a precise instrument to explain complex system behavior in a simple way, including showing in an objectively checkable and documented way the correctness of what has been built. Here *correctness* refers to a concept of high *practical* relevance, meaning that the components do what they are supposed to do and that they interact in the desired way to realize the system.

It is important for the practicability and the wide range of applicability of the ASM modeling method that the models, their refinements and their properties come in rigorous form but are not formalized in a (necessarily restricted) programming or logic language. In fact, 'modeling is not tied to computers' [134]. Nevertheless, where needed, ASMs can be further supported by formally defined executable versions and mechanically supported forms of logical verification. We adhere to the **principle to be as precise as appropriate for the given problem** domain, as detailed as needed to avoid ambiguity and incompleteness but not more, and to use formalization only where unavoidable (e. g. to refine a model to an executable or mechanically verifiable one). There is no absolute notion of accuracy. We avoid in this book any particular formal (programming or logic) description language or

logical proof system. Instead we use only basic set-theoretic and algorithmic notation and rigorous reasoning the practitioner works with on an every-day basis when explaining design ideas. The wide-ranging flexibility this yields for mapping ASM models and reasonings about their properties to a variety of logic languages and mechanically supported verification techniques is not the subject of this (but could be the subject for another valuable) book. Meantime the reader who is interested in the theoretical developments around the concept of ASMs, in particular on verification issues (including a logic for ASMs), is referred to [76, Chaps. 7,8]. Also various non-trivial mathematical proofs of behavioral properties for theoretically challenging ASMs can be found there (or in [232]).

How to Use the Book

This book is written for self-study and to serve as reference book, but it can also be used for teaching. Exercises are inserted to help readers to check their understanding of the explained concepts. For many models defined in this book refinements to executable versions can be downloaded for experimental validation from the website of the book

http://modelingbook.informatik.uni-ulm.de

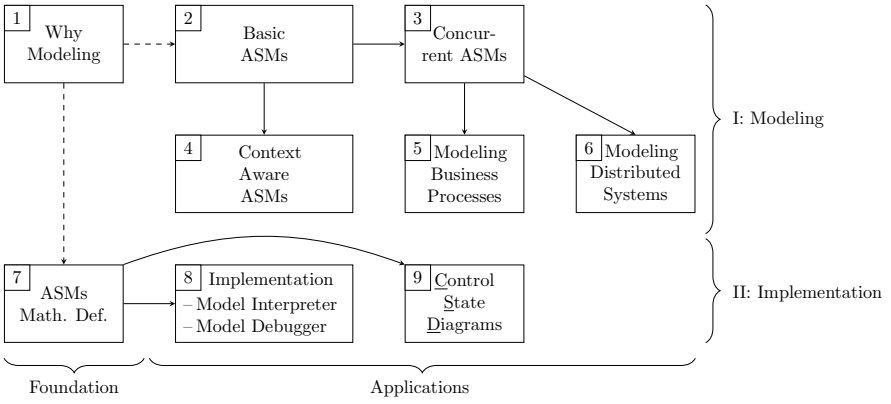
We deposit there also hints and solutions for some of the exercises and more examples. As support for teaching we also make slides on various themes treated in the book freely available; an acknowledgment would be appreciated where the material is used. In the two introductory chapters 2, 3 we have made a particular effort to start from scratch and to explain each concept separately. As a consequence, the reader who wants to learn what ASMs are and how to use them is advised to read these two chapters sequentially, from beginning to end. The remaining chapters can be read selectively. To facilitate this, we add a summary of some notational conventions and a list of frequently appearing symbols and abbreviations.

We make extensive use of footnotes so that the main explanation flow is not disrupted by side remarks and references.

Dependency of Chapters

Chapter 1 explains **why modeling** cannot be achieved by programming alone and in which respects the two activities are methodologically complementary to each other. The chapter is of epistemological character and can be skipped by the reader who is only interested in technical details.

Fig. 1 Chapter Dependency Graph



Chapter 2 explains single-agent **basic ASMs**. Technically speaking, it depends on nothing; for its pragmatical motivation it depends on Chapter 1. Chapter 3 explains multi-agent **concurrent ASMs**. It depends on an understanding of the seven basic ASM constructs (which are recapped in Sect. 2.5).

The remaining chapters of Part I can be read independently of each other.

Chapter 4 introduces context-aware ASMs, called **ambient ASMs**, and applies them to model information hiding, programming and communication patterns. It depends on an understanding of basic ASMs (as recapped in Sect. 2.5).

Chapter 5 defines an ASM pattern for modeling web services and a domain specific class of ASM nets, which is tailored for and applied to **model business processes** and their workflows.

Chapter 6 is focussed on **modeling distributed systems**, explained by two characteristic examples: dynamic routing in ad hoc networks and distributed relaxed shared memory management.

Chapters 5, 6 assume an understanding of Definitions 2, 4 (concurrent communicating ASMs) and of concurrent ASM runs (Definition 6).

The chapters of Part II have a technical character and have been written for readers with interest in implementation issues.

Chapter 7 provides a precise, **mathematical definition** (not a formalized one) of the syntax and semantics of ASMs. Technically, this chapter does not depend on any other chapter, but it builds upon the motivations explained in Chap. 1 and Chap. 2. It permits the reader to quickly consult a rigorous, unambiguous, mathematical definition of the few fundamental ASM concepts, should the need be felt when reading the rigorous natural language explanations in the main text.

Chapter 8 defines an ASM model for **CoreASM**, a tool to execute ASMs, together with a specification of a **model debugger** component. It depends

on an understanding of the seven basic ASM constructs (which are recapped in Sect. 2.5 and detailed in Chap. 7).

Chapter 9 experiments with an ASM-based metamodeling approach to define the language of **Control State Diagrams** (CSDs), the graphical notation we use throughout the book to visualize the behavior of control state ASMs. The behavioral meaning of CSDs is defined by a translator ASM which translates CSDs to basic ASMs.

The dependency of the chapters is visualized by Fig. 1.

Naming Convention

Let the meaning choose the word, and not the other way about.
George Orwell[3]

To ease the understanding of models and to facilitate their inspection (see Sect. 1.1.1), we use the following naming convention. To denote objects of some domain and their properties, we use their natural language names (or evocative abbreviations thereof) as formal names, for example

the *color* of a given *table* is *green*

is written as:

$$color(table) = green$$

where the used formal terms *color*, *table*, and *green*, stand for the following:

$table \in Table$ -- *Table* denotes the set of tables
$green \in Color$ -- *Color* denotes the set of colors
$color\colon Table \to Color$ -- *color* denotes the associating function

We treat the same way the predicative version of the statement, 'the table is green' or 'the attribute Green is true for table'. Here 'Green' is considered as attribute which may or may not hold, also called a *Predicate* of *table*. It is written as usual $Green(table)$ (read: 'Green' holds for 'table') or $Green(table) = true$, interpreting predicates as Boolean-valued functions:

$Green\colon Table \to \{true, false\}$ -- Predicates/Sets start with capital letter

In other words, we choose names whose intuitive meaning suggests their intended interpretation and use them to accurately paraphrase the informal descriptions to-be-modeled.

[3] See [198].

Acknowledgements

This is the place to express our thanks.

We thank the Abstract State Machines (ASM) community which developed the modeling method explained in this book and since the publication of the AsmBook [76] improved it considerably. In particular we thank those who have helped with detailed critical comments on draft chapters of this book: Paolo Arcaini, Donatella Barnocchi, Don Batory, Alessandro Bianchi, Paolo Dini, Roozbeh Farahbod, Albert Fleischmann, Vincenzo Gervasi, Uwe Glässer, Henri Habrias, Andreas Hense, Felix Kossak, Markus Leitz, Alexei Lisitsa, Klaus-Dieter Schewe, Jacopo Soldani, Kirsten Winter, Simone Zenzaro.

Last but not least we thank the Alexander von Humboldt Foundation, for a grant which supported two stays of the first author at the University of Ulm for our work on the book, and Alfred Hofmann and Ralf Gerstner from Springer for their trust and patience.

Egon Börger
Alexander Raschke
Pisa and Ulm, January 2018

Contents

Symbols and Notations

Notations from Logic

not ($\neg$), **and** ($\wedge$), **or** ($\vee$) -- negation, conjunction, disjunction

iff ($\Leftrightarrow$) -- logical equivalence: if and only if

forall ($\forall$) -- universal quantifier

forsome ($\exists$), **thereissome** ($\exists$) -- existential quantifier

thereisno ($\neg\exists$) -- negated existential quantifier

$eval(exp, \mathcal{A}, \zeta)$ -- value of exp in state $\mathcal{A}$ with variable assignment ζ

$eval^{\mathfrak{A}}_{\zeta}(exp)$, $[\![exp]\!]^{\mathfrak{A}}_{\zeta}$ -- other notations for $eval(exp, \mathcal{A}, \zeta)$

$eval^{\mathfrak{A}}_{\zeta,env}$, $[\![exp]\!]^{\mathfrak{A}}_{\zeta,env}$ -- extension of $eval$ by an environment, Def. 8

$exp(x/t)$ -- result of replacing each free occurrence of x in exp by t

Set/Multiset/List Notation

NAT -- set of natural numbers $0, 1, 2, \ldots$

$|A|$ -- cardinality of A

$x \in A$ -- x is an element of A

$A \setminus B = \{a \in A \mid a \notin B\}$ -- difference set

$A \cap B$, $A \cup B$, $A \times B$ -- intersection, union, cross product

$A \subseteq B$ -- A is a subset of B

$\wp(X) = \{Y \mid Y \subseteq X\}$ -- Power set of X

$\{x \in X \mid P(x)\}$ -- set of all elements of X which satisfy P

$f\colon A \to B$ -- function f from domain A to range B

$f[a \mapsto b]$ denotes f' **where** $f = f'$ except for $f'(a) = b$

$f^b_a = f[a \mapsto b]$ -- equivalent notation, common in logic

$f(\_) = constant$ -- abbreviates **forall** x $f(x) = constant$

$\epsilon x(P(x))$ -- some x which satisfies P (Hilbert's choice operator)

$\iota x(P(x))$ -- the unique x that satisfies P (Hilbert's ι operator)

$\{\!\{\,\}\!\}$ -- empty multiset/bag

$\{\!\{p_1, p_1, p_2, \ldots, p_n\}\!\}$ -- a multiset of $n+1$ elements

$[\,]$ -- empty list (or sequence, stream, queue)

$[p_1, \ldots, p_n]$ -- list etc. of n elements, in the given order

$List(Domain)$ -- a list etc. of elements of $Domain$

$head(a)$ -- the first element of a list etc. a
$tail(a)$ -- the list etc. a, except its first element
$concatenate([p_1, \ldots, p_n], [q_1, \ldots, q_m]) = [p_1, \ldots, p_n, q_1, \ldots, q_m]$
$a < b$ -- order relation: a comes before b
$a > b$ -- order relation: a comes after b

ASM Notation. See the definitions in Sect. 7

$Upd(P, \mathcal{A}, env)$ -- update set P computes, given $\mathcal{A}, env$, see Def. 21
$\Delta(P, \mathcal{A}, env)$ -- another notation for $Upd(P, \mathcal{A}, env)$
$Locs(U)$ -- set of locations of updates in an update set U
$\mathfrak{A} + U$ -- sequel state, result of applying to state $\mathfrak{A}$ the updates in U
$S \Rightarrow_P S'$ -- P can make a move from state S to S', See Def. 22
one of $(Rules)$ -- abbreviates **choose** $R \in Rules$ **do** R
SpontaneouslyDo$(\mathcal{M})$ -- see Sect. 3.2.2
import -- see Sect. 8.1.1
stepwise$(P_1, \ldots, P_n)$ -- non-atomic sequence operator, see Sect. 4.5.1
P **seq** Q -- atomic sequence operator, see Sect. 4.5.2
$U \oplus V$ -- sequential update set merge, defined p. 171
undef -- see p. 49
$Instances(P)$ -- set of instances of an ASM P, see Def. 3
Copy$(f_1, \ldots, f_n,$ **from** x **to** $x') =$
  forall $1 \leq i \leq n$ **do** $f_i(x') := f_i(x)$

OtherNotations
Display -- output
$RT, RT(a)$ -- Route Table (of agent a)

List of Figures