JFP **19** (2): 253–261, 2009.
Printed in the United Kingdom

Book reviews

Software Abstractions: Logic, Language, and Analysis by Daniel Jackson, The MIT Press, 2006, 366pp, ISBN 978-0262101141.
doi:10.1017/S0956796808006977

Daniel Jackson's Alloy project operates in a context where software requirements and designs are frequently ill-specified or bogged down by a myriad of implementation details. The aim of the project is to provide tools and analysis methods that software designers can use to "bring them back to thinking deeply about underlying concepts." The project is achieving its goals and this fine book is an extremely good way for developers, students, and researchers to learn about it.

After a short introductory chapter presenting a whirlwind tour of Alloy and its use, the book settles into a structure foreshadowed by its subtitle: *Logic, Language, and Analysis*. First, the relational logic used by Alloy is presented in detail. Alloy models software behavior using signatures populated by un-interpreted atoms, and first-order relations between these signatures. A rich collection of operators enables relations to be defined easily. Unlike some similar systems, Alloy does not distinguish scalars and sets from general relations, representing them as singleton and unary relations, respectively. This choice makes the logic more uniform and eliminates explicit handling of some corner cases.

The Alloy language, described next, embeds the logic within constructs for defining signatures and typed relations between them. Supplementing signature declarations are various forms of constraints: facts (constraints that always hold), predicates (reusable constraints), functions (reusable expressions), and assertions (implications to be checked). Commands allow assertions to be checked or predicates to be run.

The last main chapter describes the analysis that can be performed on an Alloy model. In brief, the model is converted into a Boolean satisfiability problem that is passed to a separate constraint solver. Solutions found by the solver are mapped back to the model for presentation to the user as very accessible graphical depictions of the relation instances. For instance, violations of assertions are illustrated using counter-examples.

The most controversial aspect of the analysis is the finite scope under which it occurs. Alloy commands specify the maximum number of atoms in each signature, since most models are infinite and an exhaustive search is not possible. Alloy operates under a so-called *small state hypothesis*, which asserts that most flaws in models can be illustrated by small instances. This view takes some getting used to, but the book returns to this point frequently and presents many examples to illustrate the effectiveness of the analysis that is possible.

A notable feature of the book is its comprehensive coverage. Apart from the main chapters already discussed, it contains a lengthy chapter of nontrivial examples, and over a hundred pages of appendices containing exercises, a complete reference manual, the semantics of the kernel language, and, most interestingly, a review of alternative approaches to software specification containing models contributed by the developers of the other approaches. The presentation is very accessible, particularly due to the conversational style of the questions and answers included in most of the main sections.

Since the book was published, the Alloy system has been updated to version 4 from version 3 that is described in the book. The relatively small number of user-visible differences between the systems are well summarized on the Alloy website, including the updates necessary for the examples in the book. Thus, the book is easily used with version 4.

In summary, I highly recommend *Software Abstractions: Logic, Language, and Analysis* to anyone with an interest in modeling and analyzing software. It is suitable for both class-room use and for reference long after the basics have been mastered. Systems like Alloy should be in the toolbox of all software designers and developers, so such a comprehensive book on this topic is very welcome.

Anthony M. Sloane
Macquarie University, Sydney, Australia

Peter Van Roy and Seif Haridi. *Concepts, Techniques, and Models of Computer Programming*. The MIT Press, 2004. ISBN: 0262220695 Price $70. 930pp.
doi:10.1017/S0956796808007028

I came to CTM, as this book is familiarly known, with a deep appreciation for the innovative contribution Peter Van Roy made towards implementing logic programming systems in the early 1990s. It is good to see him and his collaborators continue to push the frontiers of this tradition, and making that work accessible to the masses through this book.

CTM is of similar stock to such rigorous introductory textbooks as the classics by Abelson and Sussman (1996) and Bird and Wadler (1988), and is significantly less formal than any of Dijkstra's classics (1976). In contrast to these texts, the main theme of the book is concurrency from a systems engineering perspective, culminating in discussions of three application domains: graphical user interfaces, distributed programming and constraint programming. The reader is expected to have a reasonable grasp of the basic techniques of sequential programming, and so this book complements most other in-depth programming texts.

The target audience, late-undergraduate or early-postgraduate students, may find some of the introductory material a bit patronising. It is unfortunate that while the book is substantially about concurrency, it is itself mostly sequential-access: the reader will find it necessary to carefully peruse these early sections in order to grasp the syntax and semantics of Mozart/Oz, the programming language at the centre of the CTM world view. Some of this tedium is alleviated by the delightful ease of experimenting with the mature Mozart implementation.

Formal operational semantics are provided for the various 'kernel languages' that are used to explain language features, ultimately collected and distilled in the relatively technical Chapter 13. Readers of TaPL (Pierce, 2002) will be familiar with this approach, although here the semantics is given in the style of a concurrent constraint language (Saraswat, 1993). By itself it would be difficult to credit these sections as a sufficiently broad introduction to programming language semantics, for no properties are established. Also it is unfortunate that the Hoare logic so clearly presented in Chapter 6 is not formally related to the ongoing operational story.

At the core of the Oz approach is the *dataflow variable* (also known as the *declarative variable*), an object that can be declared in one scope and bound in another. Prolog programmers will be on familiar ground with their use in difference structures (Section 3.4.4), and in the underpinnings of the declarative concurrency development (Section 4.3). While this style of concurrency requires linguistic support to be completely natural, there are library-based implementations in various languages that embody the abstraction.

CTM has the clearest presentation of declarative programming (broadly taken) that I have yet found; the benefits for program structure and reasoning are strongly articulated and beautifully illustrated, and the limitations are carefully teased out. The presentation of declarative concurrency is a highlight of the book, and as the authors observe, deserves to be much more widely understood and applied. To a functional programmer it is somewhat