Specifying Concurrent Systems with TLA⁺

Leslie Lamport

1 Jul 1999

Incomplete Preliminary Draft

Be sure to read the description of this document on page 3 of the Introduction.
Contents

Introduction .......................................................... 1

I Getting Started ....................................................... 5
1 A Little Simple Math ................................................ 9
   1.1 Propositional Logic ........................................... 9
   1.2 Sets .......................................................... 11
   1.3 Predicate Logic .............................................. 12

2 Specifying a Simple Clock ......................................... 15
   2.1 Behaviors ..................................................... 15
   2.2 An Hour Clock ............................................... 15
   2.3 A Closer Look at the Hour-Clock Specification .......... 18
   2.4 The Hour-Clock Specification in TLA+ ....................... 19
   2.5 Another Way to Specify the Hour Clock ................. 21

3 An Asynchronous Interface ..................................... 23
   3.1 The First Specification ...................................... 24
   3.2 Another Specification ...................................... 28
   3.3 Types: A Reminder .......................................... 30
   3.4 Definitions .................................................. 31
   3.5 Comments ..................................................... 32

4 A FIFO .............................................................. 35
   4.1 The Inner Specification ...................................... 35
   4.2 Instantiation Examined ...................................... 37
      4.2.1 Instantiation is Substitution ......................... 37
      4.2.2 Parameterized Instantiation ....................... 39
      4.2.3 Implicit Substitutions ................................. 39
      4.2.4 Instantiation Without Renaming .................... 40
   4.3 Hiding the Queue ............................................. 41
   4.4 A Bounded FIFO .............................................. 41
CONTENTS

4.5 What We’re Specifying .................................. 43

5 A Caching Memory ......................................... 45
  5.1 The Memory Interface .................................. 45
  5.2 Functions .............................................. 48
  5.3 A Linearizable Memory Specification ..................... 50
  5.4 Tuples as Functions .................................... 51
  5.5 Recursive Function Definitions .......................... 53
  5.6 A Write-Through Cache .................................. 54
  5.7 Invariance ............................................. 60
  5.8 Proving Implementation .................................. 61

6 Some More Math ............................................ 65
  6.1 Sets .................................................. 65
  6.2 Silly Expressions ...................................... 67
  6.3 Recursive Function Definitions Revisited .................. 67
  6.4 Functions versus Operators ............................... 69
  6.5 Using Functions ........................................ 71
  6.6 Choose ................................................. 72

7 Writing a Specification—Some Advice ......................... 75
  7.1 Why to Specify ........................................ 75
  7.2 What to Specify ........................................ 76
  7.3 The Grain of Atomicity .................................. 76
  7.4 The Data Structures .................................... 78
  7.5 Writing the Specification ................................ 79
  7.6 Some Further Hints ..................................... 79
  7.7 When and How to Specify ................................ 81

II More Advanced Topics ..................................... 83

8 Liveness and Fairness ...................................... 85
  8.1 Temporal Formulas ..................................... 85
  8.2 Weak Fairness ......................................... 89
  8.3 Liveness for the Memory Specification ..................... 92
  8.4 Strong Fairness ........................................ 93
  8.5 Liveness for the Write-Through Cache ...................... 95
  8.6 Quantification ......................................... 96
  8.7 Temporal Logic Examined ................................. 98
    8.7.1 A Review ........................................ 98
    8.7.2 Machine Closure ................................... 98
    8.7.3 The Unimportance of Liveness ...................... 100
    8.7.4 Temporal Logic Considered Confusing ............... 100
## CONTENTS

9 Real Time (Unwritten) ................................................................. 103

10 Other Ways to Write Specifications (Unwritten) ..................... 105
  10.1 Composing Specifications ................................................... 105
  10.2 Open-System Specifications ............................................... 106
  10.3 Interface Refinement ........................................................ 106

11 Advanced Examples (Unwritten) .............................................. 109
  11.1 Sequentially Consistent Memory .......................................... 109
  11.2 The Riemann Integral ....................................................... 109

III The Tools ................................................................. 111

12 The G Parser ........................................................................ 113
  12.1 Finding an Error ................................................................ 113

13 The TLC Model Checker ....................................................... 115
  13.1 Introduction to TLC .......................................................... 115
  13.2 How TLC Works ................................................................ 122
    13.2.1 TLC Values ............................................................... 122
    13.2.2 How TLC Evaluates Expressions ................................... 123
    13.2.3 Assignment and Replacement ....................................... 125
    13.2.4 Overriding Modules .................................................... 127
    13.2.5 How TLC Computes States .......................................... 127
    13.2.6 When TLC Computes What .......................................... 128
    13.2.7 Random Simulation ...................................................... 129
  13.3 How to Use TLC ............................................................... 130
    13.3.1 Running TLC ............................................................. 130
    13.3.2 Debugging a Specification ............................................ 131
    13.3.3 The TLC Module ........................................................ 136
    13.3.4 Checking Action Invariance ......................................... 138
    13.3.5 Checking Step Simulation ............................................. 141
    13.3.6 Some Hints ................................................................ 142
  13.4 What TLC Doesn’t Do ........................................................ 145
  13.5 Future Plans ....................................................................... 146
  13.6 The Fine Print: What TLC Really Does ................................. 147
    13.6.1 TLC Values ............................................................... 147
    13.6.2 Overridden Values ....................................................... 148
    13.6.3 Expression Evaluation .................................................. 148
    13.6.4 Fingerprinting ............................................................ 157
# CONTENTS

IV The TLA$^+$ Language 159

14 The Syntax of TLA$^+$ 163
  14.1 The Simple Grammar .......................... 164
  14.1.1 BNF Grammars ............................ 164
  14.1.2 The BNF Grammar of TLA$^+$ ................. 168
  14.2 The Complete Grammar .......................... 176
     14.2.1 Precedence and Associativity ............... 176
     14.2.2 Alignment ................................ 179
     14.2.3 Comments ................................ 180
     14.2.4 Temporal Formulas ......................... 181
     14.2.5 Two Anomalies ............................ 182
  14.3 What You Type ................................ 182

15 The Operators of TLA$^+$ 185
  15.1 Constant Operators ............................. 185
     15.1.1 Boolean Operators ......................... 185
     15.1.2 The Three Interpretations of Boolean Operators .... 188
     15.1.3 Miscellaneous Constructs .................. 189
     15.1.4 The Operators of Set Theory ............... 190
     15.1.5 Functions ................................ 192
     15.1.6 Records .................................. 194
     15.1.7 Tuples .................................. 195
     15.1.8 Strings .................................. 195
     15.1.9 Numbers .................................. 196
  15.2 Nonconstant Operators ......................... 197
     15.2.1 The Meaning of a State Function .......... 197
     15.2.2 Action Operators ......................... 198
     15.2.3 Temporal Operators ....................... 198

16 The Meaning of a Module 201
  16.1 Operators and Expressions ..................... 201
     16.1.1 Orders of Operators ....................... 201
     16.1.2 $\lambda$ Expressions ...................... 202
     16.1.3 Simplifying Operator Application .......... 203
     16.1.4 Expressions .............................. 204
  16.2 Levels ........................................ 204
  16.3 Contexts ...................................... 206
  16.4 The Meaning of a $\lambda$ Expression .......... 207
  16.5 The Meaning of a Module ....................... 208
     16.5.1 Theorems and Assumptions ................. 213
     16.5.2 Submodules .............................. 213
  16.6 Finding Modules ............................... 213
  16.7 The Semantics of Instantiation ............... 214
CONTENTS

17 The Standard Modules 219
   17.1 Module Sequences ......................... 219
   17.2 Module FiniteSets .......................... 221
   17.3 Module Bags ................................. 221
   17.4 Module RealTime ............................ 222
   17.5 The Numbers Modules ........................ 224

V Appendix 229
A The ASCII Specifications 231
   A.1 The Asynchronous Interface ..................... 231
   A.2 A FIFO ........................................ 232
   A.3 A Caching Memory ................................ 233
   A.4 The Alternating Bit Protocol .................... 238

References 241
## List of Figures

2.1 The hour clock specification—typeset and ASCII versions. . . . . . 20

3.1 The First Specification of an Asynchronous Interface . . . . . . . 27
3.2 Our second specification of an asynchronous interface. . . . . . . 30
3.3 The hour clock specification with comments. . . . . . . . . . . . 33

4.1 The specification of a FIFO, with the internal variable $q$ visible. 38
4.2 Specification of a FIFO buffer of length $N$. ........................ 43

5.1 The Specification of a Memory Interface . . . . . . . . . . . 48
5.2 The internal memory specification (beginning). . . . . . . . . . . 52
5.3 The internal memory specification (end). . . . . . . . . . . . . 53
5.4 The memory specification. . . . . . . . . . . . . . . . . . . . . 53
5.5 The write-through cache specification (beginning). . . . . . . . 56
5.6 The write-through cache specification (middle). . . . . . . . . . 57
5.7 The write-through cache specification (end). . . . . . . . . . . . 58

11.1 A specification of the Riemann integral. . . . . . . . . . . . . 110

13.1 The Alternating Bit Protocol (beginning) . . . . . . . . . . . . 118
13.2 The Alternating Bit Protocol (end) . . . . . . . . . . . . . . . . 119
13.3 Module $MCAlternatingBit$. . . . . . . . . . . . . . . . . . . 120
13.4 A configuration file for module $MCAlternatingBit$. . . . . . . 121
13.5 The standard module $TLC$ . . . . . . . . . . . . . . . . . . . . 137
13.6 An invariant of the alternating bit protocol’s next-state action. . 139
13.7 A higher-level specification of the alternating bit protocol. . . . . 142
13.8 A module for checking the alternating bit protocol. . . . . . . . 142

14.1 The definition of the grammar $GSE$ for the language SE. . . . 168
14.2 The module $BNFGrammars$ . . . . . . . . . . . . . . . . . . . . 169
14.3 The precedence ranges of operators. Left-associative operators are indicated by (a). . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 178
14.4 The ASCII representations of typeset symbols. . . . . . . . . . . 183
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>14.5</td>
<td>User-definable operator symbols.</td>
<td>184</td>
</tr>
<tr>
<td>15.1</td>
<td>The constant operators.</td>
<td>186</td>
</tr>
<tr>
<td>15.2</td>
<td>Miscellaneous constructs.</td>
<td>187</td>
</tr>
<tr>
<td>15.3</td>
<td>The action operators of TLA(^+.)</td>
<td>197</td>
</tr>
<tr>
<td>15.4</td>
<td>The temporal operators of TLA(^+.)</td>
<td>197</td>
</tr>
<tr>
<td>17.1</td>
<td>The standard <em>Sequences</em> module.</td>
<td>220</td>
</tr>
<tr>
<td>17.2</td>
<td>The standard <em>FiniteSets</em> module.</td>
<td>221</td>
</tr>
<tr>
<td>17.3</td>
<td>The standard <em>Bags</em> module.</td>
<td>223</td>
</tr>
<tr>
<td>17.4</td>
<td>The <em>Peano</em> module.</td>
<td>225</td>
</tr>
<tr>
<td>17.5</td>
<td>The <em>ProtoReals</em> module (beginning).</td>
<td>226</td>
</tr>
<tr>
<td>17.6</td>
<td>The <em>ProtoReals</em> module (end).</td>
<td>227</td>
</tr>
<tr>
<td>17.7</td>
<td>The standard <em>Naturals</em> module.</td>
<td>228</td>
</tr>
<tr>
<td>17.8</td>
<td>The standard <em>Integers</em> module.</td>
<td>228</td>
</tr>
<tr>
<td>17.9</td>
<td>The standard <em>Reals</em> module.</td>
<td>228</td>
</tr>
</tbody>
</table>
Introduction

Writing is nature’s way of letting you know how sloppy your thinking is.

Guindon

Writing a specification for a system helps us understand it. It’s a good idea to understand something before you build it, so it’s a good idea to specify a system before you implement it.

Mathematics is nature’s way of letting you know how sloppy your writing is. Specifications written in an imprecise language like English are usually imprecise. In engineering, imprecision is an invitation to error. Science and engineering have adopted mathematics as a language for writing precise descriptions.

Formal mathematics is nature’s way of letting you know how sloppy your mathematics is. The mathematics written by most mathematicians and scientists is still imprecise. Most mathematics texts are precise in the small, but imprecise in the large. Each equation is a precise assertion, but you have to read the text to understand how the equations relate to one another, and what the theorems really mean. Logicians have developed ways of eliminating the words and formalizing mathematics.

Most mathematicians and computer scientists think that writing mathematics formally, without words, is tiresome. I’ve asked a number of computer scientists the following question: How long would a formal specification of the Riemann integral of elementary calculus be, assuming only arithmetic operations on real numbers. The answers I received ranged up to 50 pages. Section 11.2 shows how to do it in about 20 lines. Once you learn how, it’s easy to express ordinary mathematics in a precise, completely formal language.

To specify systems with mathematics, we must decide what kind of mathematics to use. We can specify an ordinary sequential program by describing its output as a function of its input. So, sequential programs can be specified in terms of functions. Concurrent systems are usually described in terms of their behaviors—what they do in the course of an execution. In 1977, Amir Pnueli introduced the use of temporal logic for describing such behaviors.

Temporal logic is appealing because, in principle, it allows a concurrent system to be described by a single formula. In practice, temporal logic proved to be
cumbersome. Pnueli’s temporal logic was ideal for describing some properties of systems, but awkward for others. So, it was usually combined with some more traditional way of describing systems.

In the late 1980’s, I discovered TLA, the Temporal Logic of Actions. TLA is a simple variant of Pnueli’s original logic that makes it practical to write a specification as a single formula. Most of a TLA specification consists of ordinary, nontemporal mathematics. Temporal logic plays a significant role only in describing those properties that it’s good at describing. TLA also provides a nice way to formalize the style of reasoning about systems that has proved to be most effective in practice—a style known as assertional reasoning. However, the topic of this document is specification, not proof, so I will have little to say about proofs.

TLA provides a mathematical foundation for describing concurrent systems. To write specifications, we need a complete language built atop that foundation. I initially thought that this language should be some sort of abstract programming language whose semantics would be based on TLA. I didn’t know what kind of programming language constructs would be best, so I decided to start writing specifications directly in TLA. I intended to introduce programming constructs as I needed them. To my surprise, I discovered that I didn’t need them. What I needed was a robust language for writing mathematics.

Although mathematicians have developed the science of writing formulas, they haven’t turned that science into an engineering discipline. They have developed notations for mathematics in the small, but not for mathematics in the large. The specification of a real system can be dozens or even hundreds of pages long. Mathematicians know how to write 20-line formulas, not 20-page formulas. So, I had to introduce notations for writing long formulas. What I took from programming languages were ideas for modularizing large specifications.

The language I came up with is called TLA+. I refined TLA+ in the course of writing specifications of disparate systems. But, it has changed little in the last few years. I have found TLA+ to be quite good for specifying a wide class of systems—from program interfaces (APIs) to distributed systems. It can be used to write a precise, formal description of almost any sort of discrete system. It’s especially well suited to describing asynchronous systems—that is, systems with components that do not operate in strict lock-step.

One advantage of a precise specification language is that it enables us to build tools that can help us write correct specifications. There are now at least two such tools under development: a parser and a model checker, described in Part III. The parser can catch simple errors in any TLA+ specification. The model checker can catch many more errors, but it works on a restricted class of specifications—a class that seems to include most of the specifications of interest to industry today.
The State of this Document

This document is an incomplete draft. Here is a brief road map of what there is and who should read it.

Part I

These chapters are complete and shouldn't have too many errors. They are an introduction and should be read by everyone interested in using TLA+. They explain how to specify the class of properties known as safety properties. These properties, which can be specified with almost no temporal logic, are all that most engineers will need to know about.

Part II

Only Chapter 8 has been written. It is in pretty good shape, but probably has more errors per page than the preceding chapters. Temporal logic comes to the fore in this chapter, where it is used to specify the additional class of properties known as liveness properties.

Part III

This part describes the parser and the TLC model checker. If you are reading this because you want to use TLA+, then you’ll probably want to use these tools and should read these chapters. They are preliminary and have lots of errors. Before trying to use TLC be sure to read Section 13.4 on page 145; it describes limitations of the current version of the program.

Part IV

This part has been written, but it has not been read carefully and is undoubtedly full of errors. Chapter 14 describes the precise syntax of TLA+ and includes a BNF grammar (written in TLA+). Part I should give you a good enough working knowledge of the language for most of your needs; this chapter will answer any questions you might have about the fine points of its syntax. It also contains two figures that you may want to refer to: Figure 14.4 on page 183 lists the ASCII versions of all symbols (which appear in the examples in their typeset form), and Figure 14.5 on page 184 lists all user-definable operator symbols.

Chapter 15 describes the meanings and syntax of all the built-in operators of TLA+. It may be useful as a reference manual; if you don’t understand some operator or construct of TLA+, you can try looking it up there. But the only things most readers will ever look at in this chapter are Figures 15.1 and 15.2 on pages 186 and 187, which list and briefly describe all the built-in operators.
Chapter 16 describes the semantics of TLA+. It is of interest only to mathematically sophisticated readers who want to understand exactly what a TLA+ specification means.

Chapter 17 describes the standard modules. (The description of the RealTime module is missing.) It is of interest only to the mathematically curious who wonder how traditional mathematics is expressed in TLA+.

The Appendix
The specifications that appear in the book are typeset for easy reading by humans. To be read by a tool, a specification must be written in ASCII. The appendix includes the ASCII versions of all the specifications in Part I, as well as the specifications from Chapter 13. They may help you learn how to write TLA+ specifications in ASCII.
Part I

Getting Started
A system specification consists of a lot of ordinary mathematics glued together with a little bit of temporal logic. So, most of the work in writing a precise specification consists of expressing ordinary mathematics precisely. That’s why most of the details of TLA+ are concerned with expressing ordinary mathematics.

Unfortunately, the computer science departments in many universities apparently believe that fluency in C++ is more important than a sound education in elementary mathematics. So, some readers may be unfamiliar with the mathematics needed to write specifications. Fortunately, this mathematics is quite simple. If overexposure to C++ hasn’t destroyed your ability to think logically, you should have no trouble filling any gaps in your mathematics education. You probably learned arithmetic before being exposed to C++, so I will assume you know about numbers and arithmetic operations on them.\textsuperscript{1} I will try to explain all other mathematical concepts that you need, starting in Chapter 1 with a review of some elementary math. I hope most readers will find this review completely unnecessary.

After the brief review of simple mathematics in the next section, Chapters 2 through 5 describe TLA+ with a sequence of examples. Chapter 6 explains some more about the math used in writing specifications, and Chapter 7 reviews everything and provides some advice. By the time you finish Chapter 7, you should be able to handle most of the specification problems that you are likely to encounter in ordinary engineering practice.

\textsuperscript{1}Some readers may need reminding that numbers are not strings of bits, and $2^{33} \times 2^{33}$ equals $2^{66}$, not overflow error.
Chapter 1

A Little Simple Math

1.1 Propositional Logic

Elementary algebra is the mathematics of real numbers and the operators +, −, * (multiplication), and / (division). Propositional logic is the mathematics of the two Boolean values TRUE and FALSE and the five operators whose names (and common pronunciations) are:

- ^ conjunction (and)
- \neg negation (not)
- \lor disjunction (or)
- \Rightarrow implication (implies)
- \equiv equivalence (is equivalent to)

To learn how to compute with numbers, you had to memorize addition and multiplication tables and algorithms for calculating with multidigit numbers. Propositional logic is much simpler, since there are only two values, TRUE and FALSE. To learn how to compute with these values, all you need to know are the following definitions of the five Boolean operators:

\[ \land \quad F \land G \text{ equals TRUE iff both } F \text{ and } G \text{ equal TRUE.} \]

\[ \lor \quad F \lor G \text{ equals TRUE iff } F \text{ or } G \text{ equals TRUE (or both do).} \]

\[ \neg \quad \neg F \text{ equals TRUE iff } F \text{ equals FALSE.} \]

\[ \Rightarrow \quad F \Rightarrow G \text{ equals TRUE iff } F \text{ equals FALSE or } G \text{ equals TRUE (or both).} \]

\[ \equiv \quad F \equiv G \text{ equals TRUE iff } F \text{ and } G \text{ both equal TRUE or both equal FALSE.} \]
We can also describe these operators by truth tables. This truth table for $F \Rightarrow G$ gives its value for all four combinations of truth values of $F$ and $G$:

<table>
<thead>
<tr>
<th>$F$</th>
<th>$G$</th>
<th>$F \Rightarrow G$</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE</td>
<td>TRUE</td>
<td>TRUE</td>
</tr>
<tr>
<td>TRUE</td>
<td>FALSE</td>
<td>FALSE</td>
</tr>
<tr>
<td>FALSE</td>
<td>TRUE</td>
<td>TRUE</td>
</tr>
<tr>
<td>FALSE</td>
<td>FALSE</td>
<td>TRUE</td>
</tr>
</tbody>
</table>

People are often confused about why $\Rightarrow$ means implication. In particular, they don’t understand why FALSE $\Rightarrow$ TRUE and FALSE $\Rightarrow$ FALSE should equal TRUE. The explanation is simple. We expect that if $n$ is greater than 3 then it should be greater than 1, so $n > 3$ should imply $n > 1$. Substituting 4, 2, and 0 for $n$ in the formula $(n > 3) \Rightarrow (n > 0)$ explains why we can read $F \Rightarrow G$ as $F$ implies $G$ or, equivalently, as if $F$ then $G$.

The equivalence operator $\equiv$ is equality for Booleans. We can replace $\equiv$ by $=$, but not vice versa. (We can write FALSE $= \neg$TRUE, but not $2 + 2 = 4$.) Writing $\equiv$ instead of $=$ makes it clear that the equal expressions are Booleans.\footnote{Section 15.1.2 explains a more subtle reason for using $\equiv$ instead of $=$ for equality of Boolean values.}

Formulas of propositional logic are made up of values, operators, variables, and parentheses just like those of algebra. In algebraic formulas, $*$ has higher precedence (binds more tightly) than $+$, so $x + y * z$ means $x + (y * z)$. Similarly, $\neg$ has higher precedence than $\land$ and $\lor$, which have higher precedence than $\Rightarrow$ and $\equiv$, so $\neg F \land G \Rightarrow H$ means $((\neg F) \land G) \Rightarrow H$. Other mathematical operators like $+$ and $>$ have higher precedence than the operators of propositional logic, so $n > 0 \Rightarrow n - 1 \geq 0$ means $(n > 0) \Rightarrow (n - 1 \geq 0)$. Redundant parentheses can’t hurt and often make a formula easier to read. If you have any doubt about whether parentheses are needed, use them.

The operators $\land$ and $\lor$ are associative, just like $+$ and $*$. Associativity of $+$ means that $x + (y + z)$ equals $(x + y) + z$, so we can write $x + y + z$ without parentheses. Similarly, associativity of $\land$ and $\lor$ lets us write $F \land G \land H$ or $F \lor G \lor H$.

A tautology of propositional logic is a formula like $(F \Rightarrow G) \equiv (\neg F \lor G)$ that is true for all possible truth values of its variables. One can prove all tautologies from a few simple axioms and rules. However, that would be like computing $437 + 256$ from the axioms of arithmetic. It’s much easier to verify that a simple formula is a tautology by writing its truth table—that is, by directly calculating the value of the formula for all possible truth values of its components. The formula is a tautology iff it equals TRUE for all these values. To construct the truth table for a formula, we construct the truth table for all its subformulas. For example, the following truth table shows that $(F \Rightarrow G) \equiv (\neg F \lor G)$ is indeed a
**1.2. SETS**

Propositional logic is the basis of all mathematical reasoning. It should be as familiar to you as simple algebra. Checking that \( (F \Rightarrow G) \equiv \neg F \lor G \) is a tautology should be as easy as checking that \( 2 * (x + 3 * y) \) equals \( 2 * x + 6 * y \).

Although propositional logic is simple, complex propositional formulas can get confusing. You may find yourself trying to simplify some formula and not being sure if the simplified version means the same thing as the original. There exist a number of programs for verifying propositional logic tautologies; some of them can be found and used on the World Wide Web.

1.2 Sets

Set theory is the foundation of ordinary mathematics. A set is often described as a collection of elements, but saying that a set is a collection doesn’t explain very much. The concept of set is so fundamental that we don’t try to define it. We take as undefined concepts the notion of a set and the relation \( x \in S \), where \( x \in S \) means that \( x \) is an element of \( S \). We often say *is in* instead of *is an element of*.

A set can have a finite or infinite number of elements. The set of all natural numbers \((0, 1, 2, \text{ etc.})\) is an infinite set. The set of all natural numbers less than 3 is finite, and contains the three elements 0, 1, and 2. We can write this set \( \{0, 1, 2\} \).

A set is completely determined by its elements. Two sets are equal iff they have the same elements. Thus, \( \{0, 1, 2\} \) and \( \{2, 1, 0\} \) and \( \{0, 0, 1, 2, 2\} \) are all the same set—the unique set containing the three elements 0, 1, and 2. The empty set, which we write \( \{\} \), is the unique set that has no elements.

The most common operations on sets are:

- \( \cap \) intersection
- \( \cup \) union
- \( \subseteq \) subset
- \( \setminus \) set difference

Here are their definitions and examples of their use:

- **\( S \cap T \)** The set of elements in both \( S \) and \( T \).
  \[ \{1, -1/2, 3\} \cap \{1, 2, 3, 5, 7\} = \{1, 3\} \]

- **\( S \cup T \)** The set of elements in \( S \) or \( T \) (or both).
  \[ \{1, -1/2\} \cup \{1, 5, 7\} = \{1, -1/2, 5, 7\} \]
CHAPTER 1. A LITTLE SIMPLE MATH

$S \subseteq T$ True iff every element of $S$ is an element of $T$.
{1,3} $\subseteq$ {3,2,1}

$S \setminus T$ The set of elements in $S$ that are not in $T$.
{1,−1/2,3} \{1,5,7\} = {-1/2,3}

This is all you need to know about sets before we start looking at how to specify systems. We’ll return to set theory in Section 6.1.

1.3 Predicate Logic

Once we have sets, it’s natural to say that some formula is true for all the elements of a set, or for some of the elements of a set. Predicate logic extends propositional logic with the two quantifiers:

$\forall$ universal quantification (for all)
$\exists$ existential quantification (there exists)

The formula $\forall x \in S : F$ asserts that formula $F$ is true for every element $x$ in the set $S$. For example, $\forall n \in \text{Nat} : n + 1 > n$ asserts that the formula $n + 1 > n$ is true for all elements $n$ of the set Nat of natural numbers. This formula happens to be true.

The formula $\exists x \in S : F$ asserts that formula $F$ is true for at least one element $x$ in $S$. For example, $\exists n \in \text{Nat} : n^2 = 2$ asserts that there exists a natural number $n$ whose square equals 2. This formula happens to be false.

Formula $F$ is true for all $x \in S$ iff there is no $x \in S$ for which $F$ is false, which is true iff there is no $x \in S$ for which $\neg F$ is true. Hence, the formula

(1.1) ($\exists x \in S : F$) $\equiv$ $\neg(\forall x \in S : \neg F)$

is a tautology of predicate logic.²

Since there exists no element in the empty set, the formula $\exists x \in \{\} : F$ is false for every formula $F$. By (1.1), this implies that $\forall x \in \{\} : F$ must be true for every $F$.

The quantification in the formulas $\forall x \in S : F$ and $\exists x \in S : F$ is said to be bounded, since these formulas make an assertion only about elements in the set $S$. There is also unbounded quantification. The formula $\forall x : F$ asserts that $F$ is true for all values $x$, and $\exists x : F$ asserts that $F$ is true for at least one value of $x$—a value that is not constrained to be in any particular set. Bounded and unbounded quantification are related by the following tautologies:

$(\forall x \in S : F) \equiv (\forall x : (x \in S) \Rightarrow F)$
$(\exists x \in S : F) \equiv (\exists x : (x \in S) \land F)$

²Strictly speaking, $\epsilon$ isn’t an operator of predicate logic, so this isn’t really a predicate logic tautology.
The analog of (1.1) for unbounded quantifiers is also a tautology:

$$ (\exists x : F) \equiv \neg (\forall x : \neg F) $$

Whenever possible, it is better to use bounded than unbounded quantification in a specification. This makes the specification easier for both people and tools to understand.

Logicians have rules for proving such predicate-logic tautologies, but you shouldn’t need them. You should become familiar enough with predicate logic that simple tautologies are obvious. It can help to think of $$\forall x \in S : F$$ as the conjunction of the formulas obtained by substituting all possible elements of $$S$$ for $$x$$ in $$F$$. The associativity and commutativity of conjunction then lead to the tautology:

$$ (\forall x \in S : F) \land (\forall x \in S : G) \equiv (\forall x \in S : F \land G) $$

Similarly, you can think of $$\exists x \in S : F$$ as the disjunction of formulas, so associativity and commutativity of disjunction imply:

$$ (\exists x \in S : F) \lor (\exists x \in S : G) \equiv (\exists x \in S : F \lor G) $$

for any set $$S$$ and formulas $$F$$ and $$G$$.

Mathematicians use some obvious abbreviations for nested quantifiers. For example:

$$ \forall x \in S, y \in T : F \quad \text{means} \quad \forall x \in S : (\forall y \in T : F) $$
$$ \exists w, x, y, z \in S : F \quad \text{means} \quad \exists w \in S : (\exists x \in S : (\exists y \in S : (\exists z \in S : F))) $$

In the expression $$\exists x \in S : F$$, logicians say that $$x$$ is a *bound variable* and that occurrences of $$x$$ in $$F$$ are *bound*. For example, $$n$$ is a bound variable in the formula $$\exists n \in \text{Nat} : n + 1 > n$$, and the two occurrences of $$n$$ in the subexpression $$n + 1 > n$$ are bound. A variable $$x$$ that’s not bound is said to be *free*, and occurrences of $$x$$ that are not bound are called *free* occurrences. This terminology is rather misleading. A bound variable doesn’t really occur in a formula because replacing it by some new variable doesn’t change the formula. The two formulas

$$ \exists n \in \text{Nat} : n + 1 > n \quad \exists x \in \text{Nat} : x + 1 > x $$

are equivalent. Calling $$n$$ a variable of the first formula is a bit like calling $$a$$ a variable of that formula because it appears in the name $$\text{Nat}$$. Although misleading, this terminology is common and often convenient.
Chapter 2

Specifying a Simple Clock

2.1 Behaviors

Before we try to specify a system, let’s look at how scientists do it. For centuries, they have described a system with equations that determine how its state evolves with time, where the state consists of the values of variables. For example, the state of the system comprising the earth and the moon might be described by the values of the four variables $e_{pos}$, $m_{pos}$, $e_{vel}$, and $m_{vel}$, representing the positions and velocities of the two bodies. These values are elements in a 3-dimensional space. The earth-moon system is described by equations expressing the variables’ values as functions of time and of certain constants—namely, their masses and initial positions and velocities.

A behavior of the earth-moon system consists of a function $F$ from time to states, $F(t)$ representing the state of the system at time $t$. A computer system differs from the systems traditionally studied by scientists because we can pretend that its state changes in discrete steps. So, we represent the execution of a system as a sequence of states. Formally, we define a behavior to be a sequence of states, where a state is an assignment of values to variables. We specify a system by specifying a set of possible behaviors—the ones representing a correct execution of the system.

2.2 An Hour Clock

Let’s start with a very trivial system—a digital clock that displays only the hour. To make the system completely trivial, we ignore the relation between the display and the actual time. The hour clock is then just a device whose display cycles through the values 1 through 12. Let the variable $hr$ represent the clock’s
CHAPTER 2. SPECIFYING A SIMPLE CLOCK

display. A typical behavior of the clock is the sequence

\[(hr = 11) \rightarrow (hr = 12) \rightarrow (hr = 1) \rightarrow (hr = 2) \rightarrow \cdots\]  

of states, where \([hr = 11]\) is a state in which the variable \(hr\) has the value 11. A pair of successive states, such as \([hr = 1] \rightarrow [hr = 2]\), is called a step.

To specify the hour clock, we describe all its possible behaviors. We write an initial predicate that specifies the possible initial values of \(hr\), and a next-state relation that specifies how the value of \(hr\) can change in any step.

We don’t want to specify exactly what the display reads initially; any hour will do. So, we want the initial predicate to assert that \(hr\) can have any value from 1 through 12. Let’s call the initial predicate \(HCini\). We might informally define \(HCini\) by:

\[HCini \triangleq hr \in \{1, \ldots, 12\}\]

Later, we’ll see how to write this definition formally, without the “…” that stands for the informal and so on.

The next-state relation \(HCnxt\) is a formula expressing the relation between the values of \(hr\) in the old (first) state and new (second) state of a step. We let \(hr\) represent the value of \(hr\) in the old state and \(hr'\) represent its value in the new state. (The ’ in \(hr'\) is read prime.) We want the next-state relation to assert that \(hr'\) equals \(hr + 1\) except if \(hr\) equals 12, in which case \(hr'\) should equal 1. Using an if/then/else construct with the obvious meaning, we can define \(HCnxt\) to be the next-state relation by writing:

\[HCnxt \triangleq hr' = \text{if } hr \neq 12 \text{ then } hr + 1 \text{ else } 1\]

\(HCnxt\) is an ordinary mathematical formula, except that it contains primed as well as unprimed variables. Such a formula is called an action. An action is true or false of a step. A step that satisfies the action \(HCnxt\) is called an \(HCnxt\) step.

When an \(HCnxt\) step occurs, we sometimes say that \(HCnxt\) is executed. However, it would be a mistake to take this terminology seriously. An action is a formula, and formulas aren’t executed.

We want our specification to be a single formula, not the pair of formulas \(HCini\) and \(HCnxt\). This formula must assert about a behavior that (i) its initial state satisfies \(HCini\), and (ii) each of its steps satisfies \(HCnxt\). We express (i) as the formula \(HCini\), which we interpret as a statement about behaviors to mean that the initial state satisfies \(HCini\). To express (ii), we use the temporal-logic operator \(\Box\) (pronounced box). The temporal formula \(\Box F\) asserts that formula \(F\) is always true. In particular, \(\Box HCnxt\) is the assertion that \(HCnxt\) is true for every step in the behavior. So, \(HCini \land \Box HCnxt\) is true of a behavior if the initial state satisfies \(HCini\) and every step satisfies \(HCnxt\). This formula describes all behaviors like the one in (2.1) on this page; it seems to be the specification we’re looking for.
If we considered the clock only in isolation, and never tried to relate it to another system, then this would be a fine specification. However, suppose the clock is part of a larger system—for example, the hour display of a weather station that displays the current hour and temperature. The state of the station is described by two variables: $hr$, representing the hour display, and $tmp$, representing the temperature display. Consider this behavior of the weather station:

\[
\begin{align*}
hr = 11 & \rightarrow hr = 12 \\
hr = 12 & \rightarrow hr = 12 \\
hr = 12 & \rightarrow hr = 12 \\
\end{align*}
\]

In the second and third steps, $tmp$ changes but $hr$ remains the same. These steps are not allowed by $HC_{nxt}$, which asserts that every step must increment $hr$. The formula $HC_{ini} \land \square HC_{nxt}$ does not describe the hour clock in the weather station.

A formula that describes any hour clock must allow steps that leave $hr$ unchanged—in other words, $hr' = hr$ steps. These are called stuttering steps of the clock. A specification of the hour clock should allow both $HC_{nxt}$ steps and stuttering steps. So, a step should be allowed if it is either an $HC_{nxt}$ step or a stuttering step—that is, if it is a step satisfying $HC_{nxt} \lor (hr' = hr)$. This suggests that we adopt $HC_{ini} \land \square (HC_{nxt} \lor (hr' = hr))$ as our specification.

In TLA, we let $[HC_{nxt}]_{hr}$ stand for $HC_{nxt} \land (hr' = hr)$, so we can write the formula more compactly as $HC_{ini} \land \square [HC_{nxt}]_{hr}$.

The formula $HC_{ini} \land \square [HC_{nxt}]_{hr}$ does allow stuttering steps. In fact, it allows the behavior

\[
\begin{align*}
hr = 11 & \rightarrow hr = 12 \\
hr = 12 & \rightarrow hr = 12 \\
\end{align*}
\]

that ends with an infinite sequence of stuttering steps. This behavior describes a clock whose display attains the value 12 and then keeps that value forever—in other words, a clock that stops at 12. In a like manner, we can represent a terminating execution of any system by an infinite behavior that ends with a sequence of nothing but stuttering steps. We have no need of finite behaviors (finite sequences of states), so we consider only infinite ones.

It’s natural to require that a clock does not stop, so our specification should assert that there are infinitely many nonstuttering steps. Chapter 8 explains how to express this requirement. For now, we content ourselves with clocks that may stop, and we take as our specification of an hour clock the formula $HC$ defined by

\[
HC \triangleq HC_{ini} \land \square [HC_{nxt}]_{hr}
\]
2.3 A Closer Look at the Hour-Clock Specification

A state is an assignment of values to variables, but what variables? The answer is simple: all variables. In the behavior (2.1) on page 16, \([hr = 1]\) represents some particular state that assigns the value 1 to \(hr\). It might assign the value 23 to the variable \(tmp\) and the value \(\sqrt{-17}\) to the variable \(m\_pos\). We can think of a state as representing a potential state of the entire universe. A state that assigns 1 to \(hr\) and a particular point in 3-space to \(m\_pos\) describes a state of the universe in which the hour clock reads 1 and the moon is in a particular place. A state that assigns \(\sqrt{-2}\) to \(hr\) doesn’t correspond to any state of the universe that we recognize, because the hour-clock can’t display the value \(\sqrt{-2}\). It might represent the state of the universe after a bomb fell on the clock, making its display purely imaginary.

A behavior is an infinite sequence of states—for example:

\[(2.2) \ [hr = 11] \rightarrow [hr = 77.2] \rightarrow [hr = 78.2] \rightarrow [hr = \sqrt{-2}] \rightarrow \ldots\]

A behavior describes a potential history of the universe. The behavior (2.2) doesn’t correspond to a history that we understand, because we don’t know how the clock’s display can change from 11 to 77.2. Whatever kind of history it represents is not one in which the clock is doing what it’s supposed to.

Formula \(HC\) is a temporal formula. A temporal formula is an assertion about behaviors. We say that a behavior satisfies \(HC\) iff \(HC\) is a true assertion about the behavior. Behavior (2.1) satisfies formula \(HC\). Behavior (2.2) does not, because \(HC\) asserts that every step satisfies \(HC\_{nxt}\), and the first and third steps of (2.2) don’t. (The second step, \([hr = 77.2] \rightarrow [hr = 78.2]\), does satisfy \(HC\_{nxt}\).) We regard formula \(HC\) to be the specification of an hour clock because it is satisfied by exactly those behaviors that represent histories of the universe in which the clock functions properly.

If the clock is behaving properly, then its display should be an integer from 1 through 12. So, \(hr\) should be an integer from 1 through 12 in every state of any behavior satisfying the clock’s specification, \(HC\). Formula \(HC\_{ini}\) asserts that \(hr\) is an integer from 1 through 12, and \(\Box HC\_{ini}\) asserts that \(HC\_{ini}\) is always true. So, \(\Box HC\_{ini}\) should be true for any behavior satisfying \(HC\). Another way of saying this is that \(HC\) implies \(\Box HC\_{ini}\), for any behavior. Thus, the formula \(HC \Rightarrow \Box HC\_{ini}\) should be satisfied by every behavior. A temporal formula satisfied by every behavior is called a theorem, so \(HC \Rightarrow \Box HC\_{ini}\) should be a theorem.\(^1\)

It’s easy to see that it is: \(HC\) implies that \(HC\_{ini}\) is true initially (in the first state of the behavior), and \(\Box [HC\_{nxt}]_{hr}\) implies that each step either advances \(hr\) to its proper next value or else leaves \(hr\) unchanged. We can

\(^1\)Logicians call a formula valid if it is satisfied by every behavior; they reserve the term theorem for provably valid formulas.
formalize this reasoning using the proof rules of TLA, but I’m not going to delve into proofs and proof rules.

### 2.4 The Hour-Clock Specification in TLA$^+$

Figure 2.1 on the next page shows how the hour clock specification can be written in TLA$^+$. There are two versions: the ASCII version on the bottom is the actual TLA$^+$ specification, the way you type it; the top version is typeset the way a “pretty-printer” might display it. Before trying to understand the specification, observe the relation between the two syntaxes:

- **Reserved words** that appear in small upper-case letters (like \texttt{EXTENDS}) are written in ASCII with ordinary upper-case letters.

- **When possible, symbols are represented pictorially in ASCII**—for example, $\Box$ is typed as [ ] and $\neq$ as #. (You can also type $\neq$ as /=.)

- **When there is no good ASCII representation, \TeX notation [1]** is used—for example, \( \in \) is typed as \texttt{\textbackslash in}.

A complete list of symbols and their ASCII equivalents appears in Figure 14.4 on page 183. I will usually show the typeset version of a specification; the ASCII versions of all specifications appear in the Appendix.

Now let’s look at what the specification says. It starts with

```
MODULE HourClock
```

which begins a module named \textit{HourClock}. TLA$^+$ specifications are partitioned into modules; the hour clock’s specification consists of this single module.

Arithmetic operators like + are not built into TLA$^+$, but are themselves defined in modules. (You might want to write a specification in which + means addition of matrices rather than numbers.) The usual operators on natural numbers are defined in the \textit{Naturals} module. Their definitions are incorporated into module \textit{HourClock} by the statement

```
EXTENDS Naturals
```

Every symbol that appears in a formula must either be a built-in operator of TLA$^+$, or else it must be declared or defined. The statement

```
VARIABLE hr
```

declares \textit{hr} to be a variable.

To define \textit{HCini}, we need to express the set \{1,\ldots,12\} formally, without the ellipsis “…” . We can write this set out completely as

\{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12\}
CHAPTER 2. SPECIFYING A SIMPLE CLOCK

MODULE HourClock

EXTENDS Naturals

VARIABLE hr

HCini ≡ hr ∈ (1 .. 12)

HCnxt ≡ hr' = IF hr ≠ 12 THEN hr + 1 ELSE 1

HC ≡ HCini ∧ □[HCnxt]hr

THEOREM HC ⇒ □HCini

--- MODULE HourClock ---

EXTENDS Naturals

VARIABLE hr

HCini == hr \in (1 .. 12)

HCnxt == hr' = IF hr # 12 THEN hr + 1 ELSE 1

HC == HCini \[ HCnxt \]_hr

THEOREM HC ⇒ []HCini

Figure 2.1: The hour clock specification—typeset and ASCII versions.

but that’s tiresome. Instead, we use the operator .. defined in the Naturals modules to write this set as 1 .. 12. In general i .. j is the set of integers from i through j, for any integers i and j. (It equals the empty set if j < i.) It’s now obvious how to write the definition of HCini. The definitions of HCnxt and HC are written just as before. (The ordinary mathematical operators of logic and set theory, like ∧ and ∈, are built into TLA+.)

The line

can appear anywhere between statements; it’s purely cosmetic and has no meaning. Following it is the statement

THEOREM HC ⇒ □HCini

of the theorem that was discussed above. This statement asserts that the formula HC ⇒ □HCini is true in the context of the statement. More precisely, it asserts that the formula follows logically from the definitions in this module, the definitions in the Naturals module, and the rules of TLA+. If the formula were not true, then the module would be incorrect.

The module is terminated by the symbol
2.5. ANOTHER WAY TO SPECIFY THE HOUR CLOCK

The specification of the hour clock is the definition of $HC$, including the definitions of the formulas $HC_{nxt}$ and $HC_{ini}$ and of the operators $\cdot$ and $+$ that appear in the definition of $HC$. Formally, nothing in the module tells us that $HC$ rather than $HC_{ini}$ is the clock’s specification. TLA$^+$ is a language for writing mathematics—in particular, for writing mathematical definitions and theorems. What those definitions represent, and what significance we attach to those theorems, lies outside the scope of mathematics and therefore outside the scope of TLA$^+$. Engineering requires not just the ability to use mathematics, but the ability to understand what, if anything, the mathematics tells us about an actual system.

2.5 Another Way to Specify the Hour Clock

The Naturals module also defines the modulus operator, which we write $\%$. The formula $i \% n$, which mathematicians write $i \mod n$, is the remainder when $i$ is divided by $n$. More formally, $i \% n$ is the natural number less than $n$ satisfying $i = q \cdot n + (i \% n)$ for some natural number $q$. Let’s express this condition mathematically. The Naturals module defines $Nat$ to be the set of natural numbers, and the assertion that there exists a $q$ in the set $Nat$ satisfying a formula $F$ is written $\exists q \in Nat : F$. Thus, if $i$ and $n$ are elements of $Nat$ and $n > 0$, then $i \% n$ is the unique number satisfying

$$(i \% n \leq (n - 1)) \land (\exists q \in Nat : i = q \cdot n + (i \% n))$$

We can use $\%$ to simplify our hour-clock specification a bit. Observing that $(11 \% 12)+1$ equals 12 and $(12 \% 12)+1$ equals 1, we can define a different next-state action $HC_{nxt2}$ and a different formula $HC2$ to be the clock specification:

$$HC_{nxt2} \triangleq hr' = (hr \% 12) + 1 \quad HC2 \triangleq HC_{ini} \land \Box[HC_{nxt2}]_{hr}$$

Actions $HC_{nxt}$ and $HC_{nxt2}$ are not equivalent. The step $[hr = 24] \rightarrow [hr = 25]$ satisfies $HC_{nxt}$ but not $HC_{nxt2}$, while the step $[hr = 24] \rightarrow [hr = 1]$ satisfies $HC_{nxt2}$ but not $HC_{nxt}$. However, any step starting in a state with $hr$ in $1 \ldots 12$ satisfies $HC_{nxt}$ iff it satisfies $HC_{nxt2}$. It’s therefore not hard to deduce that any behavior starting in a state satisfying $HC_{ini}$ satisfies $\Box[HC_{nxt}]_{hr}$ iff it satisfies $\Box[HC_{nxt2}]_{hr}$. Hence, formulas $HC$ and $HC2$ are equivalent. It doesn’t matter which of them we take to be the specification of an hour clock.

Mathematics provides infinitely many ways of expressing the same thing. The expressions $6 + 6$, $3 \times 4$, and $141 - 129$ all have the same meaning; they are just different ways of writing the number 12. We could replace either instance of the number 12 in module HourClock by any of these expressions without changing the meaning of any of the module’s formulas.
When writing a specification, you will often be faced with a choice of how to express something. When that happens, you should first make sure that the choices yield equivalent specifications. If they do, then you can choose the one that you feel makes the specification easiest to understand. If they don’t, then you must decide which one you mean.
Chapter 3

An Asynchronous Interface

We now specify an interface for transmitting data between asynchronous devices. A sender and a receiver are connected as shown here:

![Diagram of sender and receiver with inputs and outputs](image)

Data is sent on val, and the rdy and ack lines are used for synchronization. The sender must wait for an acknowledgement (an Ack) for one data item before it can send the next. The interface uses the standard two-phase handshake protocol, described by the following sample behavior.

\[
\begin{align*}
\{ \text{val} = 26 \} & \xrightarrow{\text{Send 37}} \{ \text{val} = 37 \} \\
\text{rdy} = 0 & \quad \text{rdy} = 1 \\
\text{ack} = 0 & \quad \text{ack} = 1
\end{align*}
\]

\[
\begin{align*}
\{ \text{val} = 4 \} & \xrightarrow{\text{Ack}} \{ \text{val} = 4 \} \\
\text{rdy} = 0 & \quad \text{rdy} = 0 \\
\text{ack} = 1 & \quad \text{ack} = 0
\end{align*}
\]

\[
\begin{align*}
\{ \text{val} = 37 \} & \xrightarrow{\text{Ack}} \{ \text{val} = 19 \} \\
\text{rdy} = 1 & \quad \text{rdy} = 1 \\
\text{ack} = 1 & \quad \text{ack} = 0
\end{align*}
\]

(It doesn’t matter what value val has in the initial state.)

It’s easy to see from this sample behavior what the set of all possible behaviors should be—once we decide what the data values are that can be sent. But, before writing the TLA+ specification that describes these behaviors, let’s look at what I’ve just done.

In writing this behavior, I made the decision that val and rdy should change in a single step. The values of the variables val and rdy represent voltages
on some set of wires in the physical device. Voltages on different wires don’t change at precisely the same instant. I decided to ignore this aspect of the physical system and pretend that the values of \textit{val} and \textit{rdy} represented by those voltages change instantaneously. This simplifies the specification, but at the price of ignoring what may be an important detail of the system. In an actual implementation of the protocol, the voltage on the \textit{rdy} line shouldn’t change until the voltages on the \textit{val} lines have stabilized; but you won’t learn that from my specification. Had I wanted the specification to convey this requirement, I would have written a behavior in which the value of \textit{val} and the value of \textit{rdy} change in separate steps.

A specification is an abstraction. It describes some aspects of the system and ignores others. We want the specification to be as simple as possible, so we want to ignore as many details as we can. But, whenever we omit some aspect of the system from the specification, we admit a potential source of error. With my specification, we can verify the correctness of a system that uses this interface, and the system could still fail because the implementor didn’t know that the \textit{val} line should stabilize before the \textit{rdy} line is changed.

The hardest part of writing a specification is choosing the proper abstraction. I can teach you about TLA$^+$, so expressing an abstract view of a system as a TLA$^+$ specification becomes a straightforward task. But, I don’t know how to teach you about abstraction. A good engineer knows how to abstract the essence of a system and suppress the unimportant details when specifying and designing it. The art of abstraction is learned only through experience.

When writing a specification, you must first choose the abstraction. In a TLA$^+$ specification, this means choosing (i) the variables that represent the system’s state and (ii) the granularity of the steps that change those variables’ values. Should the \textit{rdy} and \textit{ack} lines be represented as separate variables or as a single variable? Should \textit{val} and \textit{rdy} change in one step, two steps, or an arbitrary number of steps? To help make these choices, I recommend that you start by writing the first few steps of one or two sample behaviors, just as I did at the beginning of this section. Chapter 7 has more to say about these choices.

### 3.1 The First Specification

Now let’s specify the interface with a module \textit{AsynchInterface}. The variables \textit{rdy} and \textit{ack} can assume the values 0 and 1, which are natural numbers, so our module \texttt{EXTENDS} the \texttt{Naturals} module. We next decide what the possible values of \textit{val} should be—that is, what data values may be sent. We could write a specification that places no restriction on the data values. The specification could allow the sender to first send 37, then send $\sqrt{-15}$, and then send \textit{Nat} (the entire set of natural numbers). However, any real device can send only a restricted set of values. We could pick some specific set—for example, 32-bit
numbers. However, the protocol is the same regardless of whether it’s used to send 32-bit numbers or 128-bit numbers. So, we compromise between the two extremes of allowing anything to be sent and allowing only 32-bit numbers to be sent by assuming only that there is some set of data values that may be sent. The constant Data is a parameter of the specification. It’s declared by the statement

\begin{verbatim}
CONSTANT Data
\end{verbatim}

Our three variables are declared by

\begin{verbatim}
VARIABLES val, rdy, ack
\end{verbatim}

The keywords VARIABLE and VARIABLES are synonymous, as are CONSTANT and CONSTANTS.

The variable rdy can assume any value—for example, \(-1/2\). That is, there exist states that assign the value \(-1/2\) to rdy. When discussing the specification, we usually say that rdy can assume only the values 0 and 1. What we really mean is that the value of rdy equals 0 or 1 in every state of any behavior satisfying the specification. But, a reader of the specification shouldn’t have to understand the complete specification to figure this out. We can make the specification easier to understand by telling the reader what values the variables can assume in a behavior that satisfies the specification. We could do this with comments, but I prefer to use a definition like this one:

\begin{verbatim}
TypeInvariant = (\val \in Data) \land (\rdy \in \{0,1\}) \land (\ack \in \{0,1\})
\end{verbatim}

I call the set \{0,1\} the type of rdy, and I call TypeInvariant a type invariant. Let’s define type and some other terms more precisely:

- A state function is an ordinary expression (one with no prime or \(\Box\)) that can contain variables and constants.
- A state predicate is a Boolean-valued state function.
- An invariant Inv of a specification Spec is a state predicate such that Spec \(\Rightarrow \Box Inv\) is a theorem.
- A variable \(v\) has type \(T\) in a specification Spec iff \(v \in T\) is an invariant of Spec.

We can make the definition of TypeInvariant easier to read by writing it as follows.

\begin{verbatim}
TypeInvariant \triangleq (\val \in Data) \land (\rdy \in \{0,1\}) \land (\ack \in \{0,1\})
\end{verbatim}
Each conjunct begins with a $\land$ and must lie completely to the right of that $\land$. (The conjunct may occupy multiple lines). We use a similar notation for disjunctions. When using this bulleted-list notation, the $\land$’s or $\lor$’s must line up precisely (even in the \texttt{ascii} input). Because the indentation is significant, we can eliminate parentheses, making this notation especially useful when conjunctions and disjunctions are nested.

The initial predicate is straightforward. Initially, $val$ can equal any element of $Data$. We can start with $rdy$ and $ack$ either both 0 or both 1.

$$Init \triangleq \land val \in Data$$

$$\land rdy \in \{0, 1\}$$

$$\land ack = rdy$$

Now for the next-state action $Next$. A step of the protocol either sends a value or receives a value. We define separately the two actions $Send$ and $Rcv$ that describe the sending and receiving of a value. A $Next$ step (one satisfying action $Next$) is either a $Send$ step or a $Rcv$ step, so it is a $Send \lor Rcv$ step. Therefore, $Next$ is defined to equal $Send \lor Rcv$. Let’s now define $Send$ and $Rcv$.

We say that action $Send$ is enabled in a state from which it is possible to take a $Send$ step. From the sample behavior above, we see that $Send$ is enabled iff $rdy$ equals $ack$. Usually, the first question we ask about an action is, when is it enabled? So, the definition of an action usually begins with its enabling condition. The first conjunct in the definition of $Send$ is therefore $rdy = ack$. The next conjuncts tell us what the new values of the variables $val$, $rdy$, and $ack$ are. The new value $val'$ of $val$ can be any element of $Data$—that is, any value satisfying $val' \in Data$. The value of $rdy$ changes from 0 to 1 or from 1 to 0, so $rdy'$ equals $1 - rdy$ (because $1 = 1 - 0$ and $0 = 1 - 1$). The value of $ack$ is left unchanged. TLA$^+$ defines UNCHANGED $v$ to mean that the expression $v$ has the same value in the old and new states. More precisely, UNCHANGED $v$ equals $v' = v$, where $v'$ is the expression obtained from $v$ by priming all variables. So, we define $Send$ by:

$$Send \triangleq \land rdy = ack$$

$$\land val' \in Data$$

$$\land rdy' = 1 - rdy$$

$$\land \text{UNCHANGED } ack$$

(I could have written $ack' = ack$ instead of UNCHANGED $ack$, but I prefer the UNCHANGED construct.)

A $Rcv$ step is enabled iff $rdy$ is different from $ack$; it complements the value of $ack$ and leaves $rdy$ and $ack$ unchanged. Both $rdy$ and $ack$ are left unchanged iff the pair of values $rdy$, $ack$ is left unchanged. TLA$^+$ uses angle brackets $\langle$ and $\rangle$ to enclose ordered tuples, so $Rcv$ asserts that $\langle rdy, ack \rangle$ is left unchanged. (Angle brackets are typed in \texttt{ascii} as $\langle$ and $\rangle$.) The definition of $Rcv$ is
3.1. THE FIRST SPECIFICATION

Module AsynchInterface

EXTENDS Naturals
CONSTANT Data
VARIABLES val, rdy, ack

TypeInvariant ≜ \(\forall val \in Data\)
\(\land rdy \in \{0, 1\}\)
\(\land ack \in \{0, 1\}\)

Init \(\triangleq \land val \in Data\)
\(\land rdy \in \{0, 1\}\)
\(\land ack = rdy\)

Send \(\triangleq \land rdy = ack\)
\(\land val' \in Data\)
\(\land rdy' = 1 - rdy\)
\(\land UNCHANGED\, ack\)

Rcv \(\triangleq \land rdy \neq ack\)
\(\land ack' = 1 - ack\)
\(\land UNCHANGED\, \langle val, rdy\rangle\)

Next \(\triangleq Send \lor Rcv\)

Spec \(\triangleq Init \land [\square]_{\langle val, rdy, ack\rangle} Next\)

Theorem Spec \(\Rightarrow [\square] TypeInvariant\)

Figure 3.1: The First Specification of an Asynchronous Interface

therefore:

Rcv \(\triangleq \land rdy \neq ack\)
\(\land ack' = 1 - ack\)
\(\land UNCHANGED\, \langle val, rdy\rangle\)

As in our clock example, the complete specification Spec should allow stuttering steps—in this case, ones that leave all three variables unchanged. So, Spec allows steps that leave \(\langle val, rdy, ack\rangle\) unchanged. Its definition is

Spec \(\triangleq Init \land [\square]_{\langle val, rdy, ack\rangle} Next\)

Module AsynchInterface also asserts the invariance of TypeInvariant. It appears in full in Figure 3.1 on this page.
3.2 Another Specification

Module \textit{AsynchInterface} is a fine description of the interface and its handshake protocol. However, it’s not easy to use it to help specify a system that uses the interface. Let’s rewrite the interface specification in a form that makes it more convenient to use as part of a larger specification.

The first problem with the original specification is that it uses three variables to describe a single interface. A system might use several different instances of the interface. To avoid a proliferation of variables, we replace the three variables \texttt{val}, \texttt{rdy}, \texttt{ack} with a single variable \texttt{chan} (short for \textit{channel}). A mathematician would do this by letting the value of \texttt{chan} be an ordered triple—for example, a state \([\texttt{chan} = (−1/2, 0, 1)]\) might replace the state with \(\texttt{val} = −1/2, \texttt{rdy} = 0, \text{ and } \texttt{ack} = 1\). But programmers have learned that using tuples like this leads to mistakes; it’s easy to forget if the \texttt{ack} line is represented by the second or third component. TLA\(^+\) therefore provides records in addition to more conventional mathematical notation.

Let’s represent the state of the channel as a record with \texttt{val}, \texttt{rdy}, and \texttt{ack} fields. If \(r\) is such a record, then \(r.val\) is its \texttt{val} field. The type invariant asserts that the value of \texttt{chan} is an element of the set of all such records \(r\) in which \(r.val\) is an element of the set \(\texttt{Data}\) and \(r.rdy\) and \(r.ack\) are elements of the set \(\{0, 1\}\). This set of records is written:

\[
[\texttt{val} : \texttt{Data}, \texttt{rdy} : \{0, 1\}, \texttt{ack} : \{0, 1\}]
\]

The components of a record are not ordered, so it doesn’t matter in what order we write them. This same set of records can also be written as:

\[
[\texttt{ack} : \{0, 1\}, \texttt{val} : \texttt{Data}, \texttt{rdy} : \{0, 1\}]
\]

Initially, \texttt{chan} can equal any element of this set whose \texttt{ack} and \texttt{rdy} fields are equal, so the initial predicate is the conjunction of the type invariant and the condition \(\texttt{chan.ack} = \texttt{chan.rdy}\).

A system that uses the interface may perform an operation that sends some data value \(d\) and performs some other changes that depend on the value \(d\). We’d like to represent such an operation as an action that is the conjunction of two separate actions: one that describes the sending of \(d\) and the other that describes the other changes. Thus, instead of defining an action \texttt{Send} that sends some unspecified data value, we define the action \texttt{Send(d)} that sends data value \(d\). The next-state action is satisfied by a \texttt{Send(d)} step, for some \(d\) in \(\texttt{Data}\), or a \texttt{Rcv} step. (The value received by a \texttt{Rcv} step equals \texttt{chan.val}.) Saying that a step is a \texttt{Send(d)} step for some \(d\) in \(\texttt{Data}\) means that there exists a \(d\) in \(\texttt{Data}\) such that the step satisfies \texttt{Send(d)}—in other words, that the step is an \(\exists d \in \texttt{Data} : \texttt{Send(d)}\) step. So we define

\[
\text{Next} \triangleq (\exists d \in \texttt{Data} : \texttt{Send(d)}) \lor \texttt{Rcv}
\]
3.2. ANOTHER SPECIFICATION

The \( \text{Send}(d) \) action asserts that \( \text{chan}' \) equals the record \( r \) such that:

\[
\begin{align*}
\text{r.val} &= d \\
\text{r.rdy} &= 1 - \text{chan.rdy} \\
\text{r.ack} &= \text{chan.ack}
\end{align*}
\]

This record is written in TLA\(^{+} \) as:

\[
[\text{val} \mapsto d, \ \text{rdy} \mapsto 1 - \text{chan.rdy}, \ \text{ack} \mapsto \text{chan.ack}]
\]

(The symbol \( \mapsto \) is typed in ASCII as \( \rightarrow \).) The fields of records are not ordered, so this record can just as well be written:

\[
[\text{ack} \mapsto \text{chan.ack}, \ \text{val} \mapsto d, \ \text{rdy} \mapsto 1 - \text{chan.rdy}]
\]

The enabling condition of \( \text{Send}(d) \) is that the \( \text{rdy} \) and \( \text{ack} \) lines are equal, so we can define:

\[
\begin{align*}
\text{Send}(d) & \triangleq \\
& (\text{chan.rdy} = \text{chan.ack}) \\
& (\text{chan}' = [\text{val} \mapsto d, \ \text{rdy} \mapsto 1 - \text{chan.rdy}, \ \text{ack} \mapsto \text{chan.ack}])
\end{align*}
\]

This is a perfectly good definition of \( \text{Send}(d) \). However, I prefer a slightly different one. We can describe the value of \( \text{chan}' \) by saying that it is the same as the value of \( \text{chan} \) except that its \( \text{val} \) component equals \( d \) and its \( \text{rdy} \) component equals \( 1 - \text{chan.rdy} \). In TLA\(^{+} \), we can write this value as

\[
[\text{chan} \text{ except } ! . \text{val} = d, ! \text{rdy} = 1 - \text{chan.rdy}]
\]

Think of the ! as standing for \( \text{chan} \), the record being modified by the EXCEPT expression. In the value replacing \( !.\text{rdy} \), the symbol @ stands for \( \text{chan.rdy} \), so we can write this expression as:

\[
[\text{chan} \text{ except } ! . \text{val} = d, ! . \text{rdy} = 1 - @]
\]

In general, for any record \( r \), the expression

\[
[r \text{ except } ! . c_1 = e_1, \ldots, ! . c_n = e_n]
\]

is the record obtained from \( r \) by replacing \( r.c_i \) with \( e_i \), for each \( i \) in 1 .. \( n \). An @ in the expression \( e_i \) stands for \( r.c_i \). Using this notation, we define:

\[
\begin{align*}
\text{Send}(d) & \triangleq \\
& (\text{chan.rdy} = \text{chan.ack}) \\
& (\text{chan}' = [\text{chan} \text{ except } ! . \text{val} = d, ! . \text{rdy} = 1 - @])
\end{align*}
\]

The definition of \( \text{Rcv} \) is straightforward. A value can be received when \( \text{chan.rdy} \neq \text{chan.ack} \), and receiving the value complements \( \text{chan.ack} \):

\[
\begin{align*}
\text{Rcv} & \triangleq \\
& (\text{chan.rdy} \neq \text{chan.ack}) \\
& (\text{chan}' = [\text{chan} \text{ except } ! . \text{ack} = 1 - @])
\end{align*}
\]

The complete specification appears in Figure 3.2 on the next page.
CHAPTER 3. AN ASYNCHRONOUS INTERFACE

MODULE Channel

EXTENDS Naturals
CONSTANT Data

VARIABLE chan

$\text{TypeInvariant} \triangleq \text{chan} \in \{\text{val} : \text{Data}, \text{rdy} : \{0,1\}, \text{ack} : \{0,1\}\}$

$\text{Init} \triangleq \text{TypeInvariant}$

$\land \text{chan.ack} = \text{chan.rdy}$

$\text{Send}(d) \triangleq \land \text{chan.rdy} = \text{chan.ack}$

$\land \text{chan'} = [\text{chan \text{except} !.val} = d, \text{!,rdy} = 1 - @]$

$\text{Rcv} \triangleq \land \text{chan.rdy} \neq \text{chan.ack}$

$\land \text{chan'} = [\text{chan \text{except} !.ack} = 1 - @]$ $\text{Next} \triangleq (\exists d \in \text{Data} : \text{Send}(d)) \lor \text{Rcv}$

$\text{Spec} \triangleq \text{Init} \land \square[\text{Next}, \text{chan}]$

THEOREM $\text{Spec} \Rightarrow \square \text{TypeInvariant}$

Figure 3.2: Our second specification of an asynchronous interface.

We have now written two different specifications of the asynchronous interface. They are two different mathematical representations of the same physical system. In module AsynchInterface, we represented the system with the three variables $\text{val}, \text{rdy}, \text{and} \text{ack}$. In module Channel, we used a single variable $\text{chan}$. Since these two representations are at the same level of abstraction, they should, in some sense, be equivalent. Section 5.8 explains one sense in which they’re equivalent.

3.3 Types: A Reminder

As defined in Section 3.1, a variable $v$ has type $T$ in specification $\text{Spec}$ iff $v \in T$ is an invariant of $\text{Spec}$. Thus, $hr$ has type $1 \ldots 12$ in the specification $\text{HC}$ of the hour clock. This assertion does not mean that the variable $hr$ can assume only values in the set $1 \ldots 12$. A state is an arbitrary assignment of values to variables, so there exist states in which the value of $hr$ is $-2$. The assertion does mean that, in every behavior satisfying formula $\text{HC}$, the value of $hr$ is an element of $1 \ldots 12$.

If you are used to types in programming languages, it may seem strange that TLA$^+$ allows a variable to assume any value. Why not restrict our states to ones in which variables have the values of the right type? In other words, why
not add a formal type system to TLA⁺? A complete answer would take us too far afield. The question is addressed further in Section 6.2. For now, remember that TLA⁺ is an untyped language. Type correctness is just a name for a certain invariance property. Assigning the name TypeInvariant to a formula gives it no special status.

### 3.4 Definitions

Let’s examine what a definition means. If \( \text{Id} \) is a simple identiﬁer like \( \text{Init} \) or \( \text{Spec} \), then the deﬁnition \( \text{Id} \overset{\triangle}{=} \text{exp} \) deﬁnes \( \text{Id} \) to be synonymous with the expression \( \text{exp} \). Replacing \( \text{Id} \) by \( \text{exp} \), or vice-versa, in any expression \( e \) does not change the meaning of \( e \). This replacement must be done after the expression is parsed, not in the “raw input”. For example, the deﬁnition \( x \overset{\triangle}{=} a \times b \) makes \( x \times c \) equal to \( (a \times b) \times c \), not to \( a \times b \times c \), which equals \( a \times (b \times c) \).

The deﬁnition of \( \text{Send} \) has the form \( \text{Id}(p) \overset{\triangle}{=} \text{exp} \), where \( \text{Id} \) and \( p \) are identiﬁers. For any expression \( e \), this deﬁnes \( \text{Id}(e) \) to be the expression obtained by substituting \( e \) for \( p \) in \( \text{exp} \). For example, the deﬁnition of \( \text{Send} \) in the \( \text{Channel} \) module deﬁnes \( \text{Send}(-5) \) to equal

\[
\begin{align*}
\land & \; \text{chan}.\text{rdy} = \text{chan}.\text{ack} \\
\land & \; \text{chan}' = [\text{chan} \text{ \ except } !.\text{val} = -5, !.\text{rdy} = 1 - @]
\end{align*}
\]

\( \text{Send}(e) \) is an expression, for any expression \( e \). Thus, we can write the formula \( \text{Send}(-5) \land (\text{chan}.\text{ack} = 1) \). The identiﬁer \( \text{Send} \) by itself is not an expression, and \( \text{Send} \land (\text{chan}.\text{ack} = 1) \) is not a grammatically well-formed string. It’s non-syntactic nonsense, like \( a \times b + \).

We say that \( \text{Send} \) is an operator that takes a single argument. In the obvious way, we can deﬁne operators that take more than one argument, the general form being:

\[
\text{Id}(p_1, \ldots, p_n) \overset{\triangle}{=} \text{exp}
\] (3.1)

where the \( p_i \) are distinct identiﬁers and \( \text{exp} \) is an expression. We can consider deﬁned identiﬁers like \( \text{Init} \) and \( \text{Spec} \) to be operators that take no argument, but we generally use operator to mean an operator that takes one or more arguments.

I will use the term symbol to mean an identiﬁer like \( \text{Send} \) or an operator symbol like \( + \). Every symbol that is used in a speciﬁcation must either be a built-in operator of TLA⁺ (like \( \in \)) or it must be declared or deﬁned. Every symbol declaration or deﬁnition has a scope within which the symbol may be used. The scope of a variable or constant declaration, and of a deﬁnition, is the part of the module that follows it. Thus, we can use \( \text{Init} \) in any expression that follows its deﬁnition in module \( \text{Channel} \). The statement \text{EXTENDS Naturals} extends the scope of symbols like \( + \) deﬁned in the \( \text{Naturals} \) module to the \( \text{Channel} \) module.
The operator definition (3.1) implicitly includes a declarations of the identifiers \( p_1, \ldots, p_n \) whose scope is the expression \( \text{expr} \). An expression of the form

\[
\exists v \in S : \text{expr}
\]

has a declaration of \( v \) whose scope is the expression \( \text{expr} \). Thus the identifier \( v \) has a meaning within the expression \( \text{expr} \) (but not within the expression \( S \)).

A symbol cannot be declared or defined if it already has a meaning. The expression

\[
(\exists v \in S : \text{expr}_1) \land (\exists v \in T : \text{expr}_2)
\]

is all right, because neither declaration of \( v \) lies within the scope of the other. Similarly, the two declarations of the symbol \( d \) in the \textit{Channel} module (in the definition of \textit{Send} and in the expression \( \exists d \) in the definition of \textit{Next}) have disjoint scopes. However, the expression

\[
(\exists v \in S : (\text{expr}_1 \land \exists v \in T : \text{expr}_2))
\]

is illegal because the declaration of \( v \) in the second \( \exists v \) lies inside the scope of the its declaration in the first \( \exists v \). Although conventional mathematics and programming languages allow such redeclarations, TLA\(^+\) forbids them because they can lead to confusion and errors.

### 3.5 Comments

Even simple specifications like the ones in modules \textit{AsynchInterface} and \textit{Channel} can be hard to understand from the mathematics alone. That’s why I began with an intuitive explanation of the interface. That explanation made it easier for you to understand formula \textit{Spec} in the module, which is the actual specification. Every specification should be accompanied by an informal prose explanation. The explanation may be in an accompanying document, or it may be included as comments in the specification.

Figure 3.3 on the next page shows how the hour clock’s specification in module \textit{HourClock} might be explained by comments. In the typeset version, comments are distinguished from the specification itself by the use of a different font. As shown in the figure, TLA\(^+\) provides two way of writing comments in the \texttt{ascii} version. A comment may appear anywhere enclosed between \texttt{(* and *)}. The text of the comment itself may not contain \texttt{*}, so these comments can’t be nested. An end-of-line comment is preceded by \texttt{\textbackslash *}.

A comment almost always appears on a line by itself or at the end of a line. I put a comment between \texttt{HCnxt} and \texttt{\textbackslash \textbackslash} just to show that it can be done.

To save space, I will write few comments in the example specifications. But specifications should have lots of comments. Even if there is an accompanying document describing the system, comments are needed to help the reader understand how the specification formalizes that description.
3.5. COMMENTS

MODULE HourClock

This module specifies a digital clock that displays the current hour. It ignores real
time, not specifying when the display can change.

EXTENDS Naturals

VARIABLE hr * Variable hr represents the display.

HCini \equiv hr \in (1 .. 12)  Initially, hr can have any value from 1 through 12.

HCnxt  This is a weird place for a comment.  \Delta

hr' = IF hr \neq 12 THEN hr + 1 ELSE 1

HC \equiv HCini \land \Box[HCnxt]_hr

The complete spec. It permits the clock to stop.

THEOREM HC => \Box HCini * Type-correctness of the spec.

Figure 3.3: The hour clock specification with comments.
Comments can help solve a problem posed by the logical structure of a specification. A symbol has to be declared or defined before it can be used. In module Channel, the definition of Spec has to follow the definition of Next, which has to follow the definitions of Send and Rcv. But it’s usually easiest to understand a top-down description of a system. We would probably first want to read the declarations of Data and chan, then the definition of Spec, then the definitions of Init and Next, and then the definitions of Send and Rcv. In other words, we want to read the specification more or less from bottom to top. This is easy enough to do for a module as short as Channel; it’s inconvenient for longer specifications. We can use comments to guide the reader through a longer specification. For example, we could precede the definition of Send in the Channel module with the comment:

Actions Send and Rcv below are the disjuncts of the next-state action Next.

The module structure also allows us to choose the order in which a specification is read. For example, we can rewrite the hour-clock specification by splitting the HourClock module into three separate modules:

- **HCVar** A module that declares the variable hr.
- **HCActions** A module that extends modules Naturals and HCVar and defines HChini and HCnxt.
- **HCSpec** A module that extends module HCActions, defines formula HC, and asserts the type-correctness theorem.

The extends relation implies a logical ordering of the modules: HCVar precedes HCActions, which precedes HCSpec. But the modules don’t have to be read in that order. The reader can be told to read HCVar first, then HCSpec, and finally HCActions. The instance construct introduced below in Chapter 4 provides another tool for modularizing specifications.

Splitting a tiny specification like HourClock in this way would be ludicrous. But the proper splitting of modules can help make a large specification easier to read. When writing a specification, you should decide in what order it should be read. You can then design the module structure to permit reading it in that order, when each individual module is read from beginning to end. Finally, you should ensure that the comments within each module make sense when the different modules are read in the appropriate order.
Chapter 4

A FIFO

Our next example is a FIFO buffer, a device with which a sender process transmits a sequence of values to a receiver. The sender and receiver use two channels, \textit{in} and \textit{out}, to communicate with the buffer:

![Diagram of Sender, Buffer, and Receiver channels with arrows connecting them]

Values are sent over \textit{in} and \textit{out} using the asynchronous protocol specified by the \textit{Channel} module of Figure 3.2 on page 30. The system’s specification will allow behaviors with four kinds of nonstuttering steps: \textit{Send} and \textit{Rcv} actions on both the \textit{in} channel and the \textit{out} channel.

4.1 The Inner Specification

The specification of the FIFO first \texttt{extends} modules \textit{Naturals} and \textit{Sequences}. The \textit{Sequences} module defines operations on finite sequences. We represent a finite sequence as a tuple, so the sequence of three numbers 3, 2, 1 is the triple \( (3, 2, 1) \). The sequences module defines the following operators on sequences.

\begin{itemize}
  \item \textit{Seq}(S) \quad \text{The set of all sequences of elements of the set } S. \text{ For example, } (3, 7) \text{ is an element of } \textit{Seq}(\textit{Nat}).
  \item \textit{Head}(s) \quad \text{The first element of sequence } s. \text{ For example, } \textit{Head}((3, 7)) \text{ equals 3.}
\end{itemize}
Tail(s) The tail of sequence s (all but the head of s). For example, Tail(⟨3, 7⟩) equals ⟨7⟩.

Append(s, e) The sequence obtained by appending element e to the tail of sequence s. For example, Append(⟨3, 7⟩, 3) equals ⟨3, 7, 3⟩.

s ⨿ t The sequence obtained by concatenating the sequences s and t. For example, ⟨3, 7⟩ ⨿ ⟨3⟩ equals ⟨3, 7, 3⟩. (We type ⨿ in ASCII as \o.)

Len(s) The length of sequence s. For example, Len(⟨3, 7⟩) equals 2.

The FIFO’s specification continues by declaring the constant Message, which represents the set of all messages that can be sent. It then declares the variables. There are three variables: in and out, representing the channels, and a third variable q that represents the queue of buffered messages. The value of q is the sequence of messages that have been sent by the sender but not yet received by the receiver. (Section 4.3 has more to say about this additional variable q.)

We want to use the definitions in the Channel module to specify operations on the channels in and out. This requires two instances of that module—one in which the variable chan of the Channel module is replaced with the variable in of our current module, and the other in which chan is replaced with out. In both instances, the constant Data of the Channel module is replaced with Message. We obtain the first of these instances with the statement:

\[\text{InChan} \triangleq \text{instance Channel with Data} \leftarrow \text{Message}, \text{chan} \leftarrow \text{in}\]

For every symbol σ defined in module Channel, this defines InChan!σ to have the same meaning in the current module as σ had in module Channel, except with Message substituted for Data and in substituted for chan. For example, this statement defines InChan!TypeInvariant to equal

\[\text{in} \in [\text{val} : \text{Message}, \text{rdy} : \{0, 1\}, \text{ack} : \{0, 1\}]\]

(The statement does not define InChan!Data because Data is declared, not defined, in module Channel.) We introduce our second instance of the Channel module with the analogous statement:

\[\text{OutChan} \triangleq \text{instance Channel with Data} \leftarrow \text{Message}, \text{chan} \leftarrow \text{out}\]

The initial states of the in and out channels are specified by InChan!Init and OutChan!Init. Initially, no messages have been sent or received, so q should be

\[\text{1} \text{I like to use a singular noun like Message rather than a plural like Messages for the name of a set. That way, the } \in \text{ in the expression } m \in \text{Message can be read is a. This is the same convention that most programmers use for naming types.}\]
4.2 Instantiation Examined

4.2.1 Instantiation is Substitution

Consider the definition of Next in module Channel (page 30). We can remove every defined symbol that appears in that definition by using the symbol’s definition. For example, we can eliminate the expression Send(d) by expanding the definition of Send. We can repeat then this process. For example the – that appears in the expression 1 – (obtained by expanding the definition of Send) can be eliminated by using the definition of – from the Naturals module. Continuing in this way, we eventually obtain a definition for Next in terms of only the built-in operators of TLA and the parameters Data and chan of the
EXTENDS Naturals, Sequences

CONSTANT Message

VARIABLES in, out, q

InChan = INSTANCE Channel with Data ← Message, chan ← in
OutChan = INSTANCE Channel with Data ← Message, chan ← out

Init = \( \land InChan!Init \land OutChan!Init \land q = \{\} \)

TypeInvariant = \( \land InChan!TypeInvariant \land OutChan!TypeInvariant \land q \in \text{Seq}(\text{Message}) \)

SSend(msg) = \( \land InChan!\text{Send}(msg) \land \text{UNCHANGED } \langle \text{out}, q \rangle \) Send msg on channel in.

BufRcv = \( \land InChan!\text{Rcv} \land q' = \text{Append}(q, \text{in}.\text{val}) \land \text{UNCHANGED } \text{out} \) Receive message from channel in and append it to tail of q.

BufSend = \( \land q \neq \{\} \land OutChan!\text{Send}(\text{Head}(q)) \land q' = \text{Tail}(q) \land \text{UNCHANGED } \text{in} \) Enabled only if q is nonempty. Send Head(q) on channel out and remove it from q.

RRcv = \( \land OutChan!\text{Rcv} \land \text{UNCHANGED } \langle \text{in}, q \rangle \) Receive message from channel out.

Next = \( \lor \exists \text{msg} \in \text{Message : SSend(msg)} \lor \text{BufRcv} \lor \text{BufSend} \lor \text{RRcv} \)

Spec = \( \text{Init} \land \Box[\text{Next}]_{\langle \text{in}, \text{out}, q \rangle} \)

THEOREM Spec \( \Rightarrow \Box \text{TypeInvariant} \)

Figure 4.1: The specification of a FIFO, with the internal variable q visible.
4.2. INSTANTIATION EXAMINED

Channel module. We consider this to be the “real” definition of Next in module Channel. The statement

\[ InChan \triangleq \text{instance Channel with Data} \leftarrow \text{Message, chan} \leftarrow \text{in} \]

in module InnerFIFO defines InChan!Next to be the formula obtained from this real definition of Next by substituting Message for Data and in for chan. This defines InChan!Next in terms of only the built-in operators of TLA+ and the parameters Message and in of module InnerFIFO.

Let’s now consider an arbitrary instance statement

\[ IM \triangleq \text{instance } M \text{ with } p_1 \leftarrow e_1, \ldots, p_n \leftarrow e_n \]

Let \( \sigma \) be a symbol defined in module \( M \) and let \( d \) be its “real” definition. The instance statement defines \( IM!\sigma \) to have as its real definition the expression obtained from \( d \) by replacing all instances of \( p_i \) by the expression \( e_i \), for each \( i \). The definition of \( IM!\sigma \) must contain only the parameters (declared constants and variables) of the current module, not the ones of module \( M \). Hence, the \( p_i \) must consist of all the parameters of module \( M \). The \( e_i \) must be expressions that are meaningful in the current module.

4.2.2 Parameterized Instantiation

The FIFO specification uses two instances of module Channel—one with in substituted for chan and the other with out substituted for chan. We could instead use a single parametrized instance by putting the following statement in module InnerFIFO:

\[ Chan(ch) \triangleq \text{instance Channel with Data} \leftarrow \text{Message, chan} \leftarrow ch \]

For any symbol \( \sigma \) defined in module Channel and any expression \( exp \), this defines \( Chan(exp)!\sigma \) to equal formula \( \sigma \) with Message substituted for Data and \( exp \) substituted for chan. The Rcv action on channel in could then be written \( Chan(in)!Rcv \), and the Send(msg) action on channel out could be written \( Chan(out)!Send(msg) \).

The instantiation above defines Chan!Send to be an operator with two arguments. Writing Chan(out)!Send(msg) instead of Chan!Send(out, msg) is just an idiosyncrasy of the syntax. It is no stranger than the syntax for infix operators, which makes us write \( a + b \) instead of \( +(a, b) \).

4.2.3 Implicit Substitutions

The use of Message as the name for the set of transmitted values in the FIFO specification is a bit strange, since we had just used the name Data for the
analogous set in the asynchronous channel specifications. Suppose we had used \textit{Data} in place of \textit{Message} as the constant parameter of module \textit{InnerFIFO}. The first instantiation statement would then have been

\begin{center}
\textit{InChan} \triangleright= \text{instance Channel with } \text{Data} \leftarrow \text{Data, chan} \leftarrow \text{in}
\end{center}

The substitution $\text{Data} \leftarrow \text{Data}$ indicates that the constant parameter $\text{Data}$ of the instantiated module $\text{Channel}$ is replaced with the expression $\text{Data}$ of the current module. TLA$^+$ allows us to drop any substitution of the form $\sigma \leftarrow \sigma$, for a symbol $\sigma$. So, the statement above can be written as

\begin{center}
\textit{InChan} \triangleright= \text{instance Channel with chan} \leftarrow \text{in}
\end{center}

We know there is an implied $\text{Data} \leftarrow \text{Data}$ substitution because an instance statement must have a substitution for every parameter of the instantiated module. If some parameter $p$ has no explicit substitution, then there is an implicit substitution $p \leftarrow p$. This means that the instance statement must lie within the scope of a declaration or definition of the symbol $p$.

It is quite common to instantiate a module with this kind of implicit substitution. Often, every parameter has an implicit substitution, in which case the list of explicit substitutions is empty. The \textit{with} is then omitted.

\section*{4.2.4 Instantiation Without Renaming}

So far, all the instantiations we’ve used have been with renaming. For example, the first instantiation of module $\text{Channel}$ renames the defined symbol $\text{Send}$ as $\text{InChan!Send}$. This kind of renaming is necessary if we are using multiple instances of the module, or a single parameterized instance. The two instances $\text{InChan!Init}$ and $\text{OutChan!Init}$ of $\text{Init}$ in module $\text{InnerFIFO}$ are different formulas, so they need different names.

Sometimes we need only a single instance of a module. For example, suppose we are specifying a system with only a single asynchronous channel. We then need only one instance of $\text{Channel}$, so we don’t have to rename the instantiated symbols. In that case, we can write something like

\begin{center}
\text{instance Channel with Data} \leftarrow D, \text{chan} \leftarrow x
\end{center}

This instantiates $\text{Channel}$ with no renaming, but with substitution. Thus, it defines $\text{Rcv}$ to be the formula of the same name from the $\text{Channel}$ module, except with $D$ substituted for $\text{Data}$ and $x$ substituted for $\text{chan}$. The expressions substituted for an instantiated module’s parameters must be defined. So, this \text{instance} statement must be within the scope of the definitions or declarations of $D$ and $x$. 
4.3 Hiding the Queue

Module InnerFIFO of Figure 4.1 defines $Spec$ to be $Init \land \Box[Next]...$, the sort of formula we’ve become accustomed to as a system specification. However, formula $Spec$ describes the value of variable $q$, as well as of the variables $in$ and $out$. The picture of the FIFO system I drew on page 35 shows only channels $in$ and $out$; it doesn’t show anything inside the boxes. A specification of the FIFO should describe only the values sent and received on the channels. The variable $q$, which represents what’s going on inside the box labeled Buffer, is used to specify what values are sent and received. In the final specification, it should be hidden.

In TLA, we hide a variable with the existential quantifier $\exists$ of temporal logic. The formula $\exists x : F$ is true of a behavior iff there exists some sequence of values—one in each state of the behavior—that can be assigned to the variable $x$ that will make formula $F$ true. (The meaning of $\exists$ is defined more precisely in Section 8.6.)

The obvious way to write a FIFO specification in which $q$ is hidden is with the formula $\exists q : Spec$. However, we can’t put this definition in module InnerFIFO because $q$ is already declared there, and a formula $\exists q : ...$ would redeclare it. Instead, we use a new module with a parametrized instantiation of the InnerFIFO module (see Section 4.2.2 above):

```
module FIFO

constant Message
variables in, out

Inner(q) ≜ instance InnerFIFO
Spec ≜ $\exists q : Inner(q)!Spec$
```

Observe that the instance statement is an abbreviation for

```
Inner(q) ≜ instance InnerFIFO
    with q ← q, in ← in, out ← out, Message ← Message
```

The variable parameter $q$ of module InnerFIFO is instantiated with the parameter $q$ of the definition of Inner. The other parameters of the InnerFIFO module are instantiated with the parameters of module FIFO.

4.4 A Bounded FIFO

We have specified an unbounded FIFO—a buffer that can hold an unbounded number of messages. Any real system has a finite amount of resources, so it can
CHAPTER 4. A FIFO

contain only a bounded number of in-transit messages. In many situations, we wish to abstract away the bound on resources and describe a system in terms of unbounded FIFOs. In other situations, we may care about that bound. We then want to strengthen our specification by placing a bound $N$ on the number of outstanding messages.

A specification of a bounded FIFO differs from our specification of the unbounded FIFO only in that action $BufRcv$ should be enabled only when there are fewer than $N$ messages in the buffer—that is, only when $\text{Len}(q)$ is less than $N$. It would be easy to write a complete new specification of a bounded FIFO by copying module $InnerFIFO$ and just adding the conjunct $\text{Len}(q) < N$ to the definition of $BufRcv$. But let’s use module $InnerFIFO$ as it is, rather than copying it.

The next-state action $BNext$ for the bounded FIFO is the same as the FIFO’s next-state action $Next$ except that it allows a $BufRcv$ step only if $\text{Len}(q)$ is less than $N$. In other words, $BNext$ should allow a step only if (i) it’s a $Next$ step and (ii) if it’s a $BufRcv$ step, then $\text{Len}(q) < N$ is true in the first state. In other words, $BNext$ should equal

$$Next \land (\text{BufRcv} \Rightarrow (\text{Len}(q) < N))$$

Module $BoundedFIFO$ in Figure 4.2 on the next page contains the specification. It introduces the new constant parameter $N$. It also contains the statement

$$\text{assume } (N \in \text{Nat}) \land (N > 0)$$

which asserts that, in this module, we are assuming that $N$ is a positive natural number. Such an assumption has no effect on any definitions made in the module. However, it may be taken as a hypothesis when proving any theorems asserted in the module. In other words, a module asserts that its assumptions imply its theorems. It’s a good idea to assert this kind of simple assumption about constants.

An $\text{assume}$ statement should only be used to assert assumptions about constants. The formula being assumed should not contain any variables. It might be tempting to assert type declarations as assumptions—for example, to add to module $InnerFIFO$ the assumption $q \in \text{Seq}(\text{Message})$. However, that would be wrong because it asserts that, in any state, $q$ is a sequence of messages. As we observed in Section 3.3, a state is a completely arbitrary assignment of values to variables, so there are states in which $q$ has the value $\sqrt{-17}$. Assuming that such a state doesn’t exist would lead to a logical contradiction.

You may wonder why module $BoundedFIFO$ asserts that $N$ is a positive natural, but doesn’t assert that $\text{Message}$ is a set. Similarly, why didn’t we have to specify that the constant parameter $\text{Data}$ in our asynchronous interface specifications is a set? The answer is that, in TLA$^+$, every value is a set.$^2$ A

---

$^2$TLA$^+$ is based on the mathematical formalism known as Zermelo-Fränkel set theory, also called ZF.
4.5. WHAT WE’RE SPECIFYING

I wrote above, at the beginning of this section, that we were going to specify a FIFO buffer. Formula Spec of the FIFO module actually specifies a set of behaviors, each representing a sequence of sending and receiving operations on the channels in and out. The sending operations on in are performed by the sender, and the receiving operations on out are performed by the receiver. The sender and receiver are not part of the FIFO buffer; they form its environment.

Our specification describes a system consisting of the FIFO buffer and its environment. The behaviors satisfying formula Spec of module FIFO represent those histories of the universe in which both the system and its environment behave correctly. It’s often helpful in understanding a specification to indicate explicitly which steps are system steps and which are environment steps. We can do this by defining the next-state action to be

\[ \text{Next} \triangleq \text{SysNext} \lor \text{EnvNext} \]
where $SysNext$ describes system steps and $EnvNext$ describes environment steps. For the FIFO, we have

$$
SysNext \triangleq BufRcv \lor BufSend
$$

$$
EnvNext \triangleq (\exists \, \text{msg} \in \text{Message} : SSend(\text{msg})) \lor RRcv
$$

While suggestive, this way of defining the next-state action has no formal significance. The specification $Spec$ equals $Init \land \Box[Next]$,; changing the way we structure the definition of $Next$ doesn’t change its meaning. If a behavior fails to satisfy $Spec$, nothing tells us if the system or its environment is to blame.

A formula like $Spec$, which describes the correct behavior of both the system and its environment, is called a closed-system or complete-system specification. An open-system specification is one that describes only the correct behavior of the system. A behavior satisfies an open-system specification if it represents a history in which either the system operates correctly, or it failed to operate correctly only because its environment did something wrong. Section 10.2 explains how to write open-system specifications.

Open-system specifications are philosophically more satisfying. However, closed-system specifications are a little bit easier to write, and the mathematics underlying them is simpler. So, we almost always write closed-system specifications. It’s usually quite easy to turn a closed-system specification into an open-system specification. But in practice, there’s little reason to do so.
Chapter 5

A Caching Memory

A memory system consists of a set of processors connected to a memory by some abstract interface, which we label $\text{memInt}$.

In this section we specify what the memory is supposed to do, then we specify a particular implementation of the memory using caches. We begin by specifying the memory interface, which is common to both specifications.

5.1 The Memory Interface

The asynchronous interface described in Chapter 3 uses a handshake protocol. Receipt of a data value must be acknowledged before the next data value can be sent. In the memory interface, we abstract away this kind of detail and represent both the sending of a data value and its receipt as a single step. We call it a $\text{Send}$ step if a processor is sending the value to the memory; it’s a $\text{Reply}$ step if the memory is sending to a processor. Processors do not send values to one another, and the memory sends to only one processor at a time.

We represent the state of the memory interface by the value of the variable $\text{memInt}$. A $\text{Send}$ step changes $\text{memInt}$ in some way, but we don’t want to specify exactly how. The way to leave something unspecified in a specification is to make it a parameter. For example, in the bounded FIFO of Section 4.4, we left the size of the buffer unspecified by making it a parameter $N$. We’d
therefore like to declare a parameter $Send$ so that $Send(p, d)$ describes how $memInt$ is changed by a step that represents processor $p$ sending data value $d$ to the memory. However, TLA+ provides only constant and variable parameters, not action parameters.\(^1\) So, we declare $Send$ to be a constant operator and write $Send(p, d, memInt, memInt')$ instead of $Send(p, d)$.

In TLA+, we declare $Send$ to be a constant operator that takes four arguments by writing

$$constant \ Send(p, d, miOld, miNew)$$

This means that $Send(p, d, miOld, miNew)$ is an expression, for any expressions $p$, $d$, $miOld$, and $miNew$, but it says nothing about what the value of that expression is. We want it to be a Boolean value that is true if and only if the first state of the step in which $memInt$ equals $miOld$ and the second state represents the sending by $p$ of value $d$ to the memory.\(^2\) We can assert that the value is a Boolean by the assumption:

$$\text{assume } \forall p, d, \ miOld, \ miNew : \ Send(p, d, miOld, miNew) \in \text{BOOLEAN}$$

This asserts that the formula

$$Send(p, d, miOld, miNew) \in \text{BOOLEAN}$$

is true for all values of $p$, $d$, $miOld$, and $miNew$. The built-in symbol BOOLEAN denotes the set \(\{\text{TRUE, FALSE}\}\), whose elements are the two Boolean values TRUE and FALSE.

This \texttt{assume} statement asserts formally that the value of $Send(p, d, miOld, miNew)$ is a Boolean. But the only way to assert formally what that value signifies would be to say what it actually equals—that is, to define $Send$ rather than making it a parameter. We don’t want to do that, so we just state informally what the value means. This statement is part of the intrinsically informal description of the relation between our mathematical abstraction and a physical memory system.

To allow the reader to understand the specification, we have to describe informally what $Send$ means. The \texttt{assume} statement asserting that $Send(\ldots)$ is a Boolean is then superfluous as an explanation. However, it could help tools understand the specification, so it’s a good idea to include it anyway.

\(^1\)Even if TLA+ allowed us to declare an action parameter, we would have no way to specify that a $Send(p, d)$ action constrains only $memInt$ and not other variables.

\(^2\)We expect $Send(p, d, miOld, miNew)$ to have this meaning only when $p$ is a processor and $d$ a value that $p$ is allowed to send, but we simplify the specification a bit by requiring it to be a Boolean for all values of $p$ and $d$. 

5.1. THE MEMORY INTERFACE

A specification that uses the memory interface can use the operators Send and Reply to specify how the variable memInt changes. The specification must also describe memInt’s initial value. We therefore declare a constant parameter InitMemInt that is the set of possible initial values of memInt.

We also introduce three constant parameters that are needed to describe the interface:

- Proc The set of processor identifiers. (We usually shorten processor identifier to processor when referring to an element of Proc.)
- Adr The set of memory addresses.
- Val The set of possible memory values that can be assigned to an address.

Finally, we define the values that the processors and memory send to one another over the interface. A processor sends a request to the memory. We represent a request as a record with an op field that specifies the type of request and additional fields that specify its arguments. Our simple memory allows just read and write requests. A read request has op field “Rd” and an adr field specifying the address to be read. The set of all read requests is therefore the set

\[ \text{[op : \{“Rd”\}, adr : Adr]} \]

of all records whose op field equals “Rd” (is an element of the set \{“Rd”\} whose only element is the string “Rd”) and whose adr field is an element of Adr. A write request must specify the address to be written and the value to write. It is represented by a record with op field equal to “Wr”, and with adr and val fields specifying the address and value. We define MReq, the set of all requests, to equal the union of these two sets. (Set operations, including union, are described in Section 1.2.)

The memory responds to a read request with the memory value it read. We will also have it respond to a write request; and it seems nice to let the response be different from the response to any read request. We therefore require the memory to respond to a write request by returning a value NoVal that is different from any memory value. We could declare NoVal to be a constant parameter and add the assumption NoVal \notin Val. (The symbol \notin is typed in ASCII as \notin.) But it’s best, when possible, to avoid introducing parameters. Instead, we define NoVal by:

\[ \text{NoVal} \triangleq \text{choose } v : v \notin Val \]

The expression choose x : F equals an arbitrarily chosen value x that satisfies the formula F. (If no such x exists, the expression has a completely arbitrary value.) This statement defines NoVal to be some value that is not an element of
CHAPTER 5. A CACHING MEMORY

MODULE MemoryInterface

VARIABLE memInt

CONSTANTS Send(p, d, memInt, memInt'), Reply(p, d, memInt, memInt'), InitMemInt, Proc, Adr, Val

ASSUME ∀ p, d, miOld, miNew : Send(p, d, miOld, miNew) ∈ BOOLEAN ∧ Reply(p, d, miOld, miNew) ∈ BOOLEAN

MReq ≝ [op : {"Rd"}, adr : Adr] ∪ [op : {"Wr"}, adr : Adr, val : Val]

NoVal ≝ CHOOSE v : v ∈ Val | An arbitrary value not in Val.

Figure 5.1: The Specification of a Memory Interface

Val. We have no idea what the value of NoVal is; we just know what it isn’t—namely, that it isn’t an element of Val. The choose operator is discussed in Section 6.6.

The complete memory interface specification is module MemoryInterface in Figure 5.1 on this page.

5.2 Functions

A memory assigns values to addresses. The state of the memory is therefore an assignment of elements of Val (memory values) to elements of Adr (memory addresses). In a programming language, such an assignment is called an array of type Val indexed by Adr. In mathematics, it’s called a function from Adr to Val. Before writing the memory specification, let’s look at the mathematics of functions, and how it is described in TLA+.

A function f has a domain, written DOMAIN f, and it assigns to each element x of its domain the value f[x]. (Mathematicians write this as f(x), but TLA+ uses the array notation of programming languages, with square brackets.) Two functions f and g are equal iff they have the same domain and f[x] = g[x] for all x in their domain.

The range of a function f is the set of all values of the form f[x] with x in DOMAIN f. For any sets S and T, the set of all functions whose domain equals
5.2. FUNCTIONS

S and whose range is any subset of T is written \([S \rightarrow T]\).

Ordinary mathematics does not have a convenient notation for writing an expression whose value is a function. TLA+ defines \([x \in S \mapsto e]\) to be the function \(f\) with domain \(S\) such that \(f[x] = e\) for every \(x \in S\).³ For example,

\[
succ \triangleq [n \in Nat \mapsto n + 1]
\]

defines \(succ\) to be the successor function on the natural numbers—the function with domain \(Nat\) such that \(succ[n] = n + 1\) for all \(n \in Nat\).

A record is a function whose domain is a finite set of strings. For example, a record with \(val\), \(ack\), and \(rdy\) fields is a function whose domain is the set \{“val”, “ack”, “rdy”\} consisting of the three strings “val”, “ack”, and “rdy”. The expression \(r.ack\), the \(ack\) field of a record \(r\), is an abbreviation for \(r[“ack”]\). The record

\[
[val \mapsto 42, \ ack \mapsto 1, \ rdy \mapsto 0]
\]
can be written

\[
[i \in \{“val”, “ack”, “rdy”\} \Rightarrow 
\begin{align*}
&\text{if } i = “val” \text{ then } 42 \text{ else if } i = “ack” \text{ then } 1 \text{ else } 0
\end{align*}
\]

The \textsc{except} construct for records, explained in Section 3.2, is a special case of a general \textsc{except} construct for functions, where \(![c]\) is an abbreviation for \(![“c”]\).

For any function \(f\), the expression \(f \text{ except } ![c] = e\) is the function \(\hat{f}\) that is the same as \(f\) except with \(\hat{f}[c] = e\). This function can also be written:

\[
[x \in \text{domain } f \Rightarrow \text{if } x = c \text{ then } e \text{ else } f[x]]
\]

assuming that the symbol \(x\) does not occur in any of the expressions \(f\), \(c\), and \(e\). For example, \([\text{succ except } ![42] = 86]\) is the function \(g\) that is the same as \(\text{succ}\) except that \(g[42]\) equals 86 instead of 43.

As in the \textsc{except} construct for records, the expression \(e\) in

\[
[f \text{ except } ![c] = e]
\]
can contain the symbol \(\emptyset\), where it means \(f[c]\). For example,

\[
[\text{succ except } ![42] = 2 * \emptyset] = [\text{succ except } ![42] = 2 * \text{succ}[42]]
\]

In general,

\[
[f \text{ except } ![c_1] = e_1, \ldots, ![c_n] = e_n]
\]

³Computer scientists write \(\lambda x : S.e\) to mean something very much like \(x \in S \mapsto e\). Such \(\lambda\) expressions aren’t quite the same as the functions of ordinary mathematics, so TLA+ doesn’t use that notation for writing functions.
is the function \( \hat{f} \) that is the same as \( f \) except with \( \hat{f}[c_i] = e_i \) for each \( i \). More precisely, this expression equals

\[
[\ldots [f \text{ EXCEPT } ![c_1] = e_1] \text{ EXCEPT } ![c_2] = e_2] \ldots \text{ EXCEPT } ![c_n] = e_n]$

Functions correspond to the arrays of programming languages. The domain of a function corresponds to the index set of an array. The function \([f \text{ EXCEPT } ![c] = e]\) corresponds to the array obtained from \( f \) by assigning \( e \) to \( f[c] \). A function whose range is a set of functions corresponds to an array of arrays. TLA\(^+\) defines \([f \text{ EXCEPT } ![c][d] = e]\) to be the function corresponding to the array obtained by assigning \( e \) to \( f[c][d] \). It can be written as

\[
[f \text{ EXCEPT } ![c] = [@ \text{ EXCEPT } ![d] = e]]$

The generalization to \([f \text{ EXCEPT } ![c_1] \ldots ![c_n] = e]\) for any \( n \) should be obvious. Since a record is a function, this notation can be used for records as well. TLA\(^+\) uniformly maintains the notation that \( \sigma.c \) is an abbreviation for \( \sigma[[c]] \). For example, this implies:

\[
[f \text{ EXCEPT } ![c].d = e] = [f \text{ EXCEPT } ![c] = [@ \text{ EXCEPT } !.d = e]]
\]

The TLA\(^+\) definition of records as functions makes it possible to manipulate them in ways that have no counterparts in programming languages. For example, we can define an operator \( R \) such that \( R(r, s) \) is the record obtained from \( r \) by replacing the value of each field \( c \) that is also a field of the record \( s \) with \( s.c \). In other words, for every field \( c \) of \( r \), if \( c \) is a field of \( s \) then \( R(r, s).c = s.c \); otherwise \( R(r, s).c = r.c \). The definition is:

\[
R(r, s) \triangleq [c \in \text{DOMAIN } r \mapsto \text{IF } c \in \text{DOMAIN } s \text{ THEN } s[c] \text{ ELSE } r[c]]
\]

So far, I have described only functions of a single argument. TLA\(^+\) also allows functions of multiple arguments. Section 15.1.5 on page 192 describes the general versions of the TLA\(^+\) function constructs for functions with any number of arguments. However, functions of a single argument are all you really need. You can always replace a function of multiple arguments with a function-valued function—for example, writing \( f[a][b] \) instead of \( f[a, b] \).

### 5.3 A Linearizable Memory Specification

We specify a very simple memory system in which a processor \( p \) issues a memory request and then waits for a response before issuing the next request. In our specification, the request is executed by accessing (reading or modifying) a variable \( \text{mem} \), which represents the current state of the memory. Because the memory can receive requests from other processors before responding to processor \( p \), it matters when \( \text{mem} \) is accessed. We let the access of \( \text{mem} \) occur
any time between the request and the response. This specifies what is called
a linearizable memory. A less restrictive, more common type of memory called
sequentially consistent memory is specified in Section 11.1.

In addition to mem, the specification has the internal variables ctl and buf,
where ctl[p] describes the status of processor p’s request and buf[p] contains
either the request or the response. Consider the request req that equals
\[ op \mapsto \text{“Wr”}, \; \text{adr} \mapsto a, \; \text{val} \mapsto v \]

It is a request to write v to memory address a, and it generates the response
NoVal. The processing of this request is represented by the following three steps:

\[
\begin{align*}
\text{ctl}[p] & = \text{“rdy”} \quad \text{Req}(p) \quad \text{ctl}[p] = \text{“busy”} \\
\text{buf}[p] & = \ldots \\
\text{mem}[a] & = \ldots
\end{align*}
\]

\[
\begin{align*}
\text{Do}(p) \quad \text{ctl}[p] & = \text{“done”} \\
\text{buf}[p] & = \text{NoVal} \\
\text{mem}[a] & = v
\end{align*}
\]

\[
\begin{align*}
\text{Rsp}(p) \quad \text{ctl}[p] & = \text{“rdy”} \\
\text{buf}[p] & = \text{NoVal} \\
\text{mem}[a] & = v
\end{align*}
\]

A Req(p) step represents the issuing of a request by processor p. It is enabled
when ctl[p] = “rdy”; it sets ctl[p] to “busy” and sets buf[p] to the request. A
Do(p) step represents the memory access; it is enabled when ctl[p] = “busy”
and it sets ctl[p] to “done” and buf[p] to the response. A Rsp(p) step represents
the memory’s response to p; it is enabled when ctl[p] = “done” and it sets ctl[p]
to “rdy”.

Writing the specification is a straightforward exercise in representing these
changes to the variables in TLA+ notation. The internal specification, with
mem, ctl, and buf visible (free variables), appears in module InternalMemory
on the following two pages. The memory specification, which hides the three
internal variables, is module Memory in Figure 5.4 on page 53.

5.4 Tuples as Functions

Before writing our caching memory specification, let’s take a closer look at tuples.
Recall that \( \langle a, b, c \rangle \) is the 3-tuple with components a, b, and c. In TLA+,
this 3-tuple is actually the function with domain \{1, 2, 3\} that maps 1 to a, 2 to
b, and 3 to c. Thus, \( \langle a, b, c \rangle[2] \) equals b.

TLA+ provides the Cartesian product operator \( \times \) of ordinary mathematics,
where \( A \times B \times C \) is the set of all 3-tuples \( \langle a, b, c \rangle \) such that a \( \in A \),
b \( \in B \), and c \( \in C \). Note that \( A \times B \times C \) is different from \( A \times (B \times C) \),
which is the set of pairs \( \langle a, p \rangle \) with a \( \in A \) and p in the set of pairs \( B \times C \).

The Sequences module defines finite sequences to be tuples. Hence, a se-
quence of length n is a function with domain 1..n. In fact, s is a sequence iff
EXTENDS MemoryInterface
VARIABLES mem, ctl, buf

IInit $\triangleq$ The initial predicate
$\land$ mem $\in [\text{Adr} \to \text{Val}]$
$\land$ ctl $\in [\text{Proc} \to \{\text{"rdy"}, \text{"busy"}, \text{"done"}\}]
$\land$ buf $\in [\text{Proc} \to \text{MReq} \cup \text{Val} \cup \{\text{NoVal}\}]

TypeInvariant $\triangleq$ The type-correctness invariant.
$\land$ mem $\in [\text{Adr} \to \text{Val}]
$\land$ ctl $\in [\text{Proc} \to \{\text{"rdy"}, \text{"busy"}, \text{"done"}\}]
$\land$ buf $\in [\text{Proc} \to \text{MReq} \cup \text{Val} \cup \{\text{NoVal}\}]

\text{Req}(p) \triangleq$ Processor $p$ issues a request.
$\land$ ctl$[p]$ $=$ “rdy” Enabled iff $p$ is ready to issue a request.
$\land$ $\exists$ req $\in$ MReq : For some request req:
$\land$ Send$(p, \text{req}, \text{memInt}, \text{memInt}')$
$\land$ buf$'$ $=$ [buf EXCEPT ![p] $=$ req]
$\land$ ctl$'$ $=$ [ctl EXCEPT ![p] $=$ “busy”]
$\land$ UNCHANGED mem

Do(p) $\triangleq$ Perform p’s request to memory.
$\land$ ctl$[p]$ $=$ “busy” Enabled iff p’s request is pending.
ELSE mem Leave mem unchanged on a “Rd” request.
$\land$ buf$'$ $=$ [buf EXCEPT ![p] $=$ IF buf$[p].op$ $=$ “Wr” THEN NoVal ELSE mem(buf$[p].adr$)]
$\land$ ctl$'$ $=$ [ctl EXCEPT ![p] $=$ “done”]
$\land$ UNCHANGED memInt

Figure 5.2: The internal memory specification (beginning).
5.5. RECURSIVE FUNCTION DEFINITIONS

We need one more tool to write the caching memory specification: recursive function definitions. Recursively defined functions are familiar to programmers. The classic example is the factorial function, which I’ll call fact. It’s usually defined by writing:

\[
\text{fact}[n] = \begin{cases} 1 & \text{if } n = 0 \\ n \times \text{fact}[n-1] & \text{else} \end{cases}
\]

for all \( n \in \text{Nat} \). The TLA+ notation for writing functions suggests trying to define fact by

\[
\text{fact} \triangleq [n \in \text{Nat} \mapsto \begin{cases} 1 & \text{if } n = 0 \\ n \times \text{fact}[n-1] & \text{else} \end{cases}]
\]

5.5 Recursive Function Definitions

Below are a few operator definitions from the Sequences module. (The meanings of the operators are described in Section 4.1.)

- Head \( s \triangleq s[1] \)
- Tail \( s \triangleq [i \in 1 .. (\text{Len}(s) - 1) \mapsto s[i+1]] \)
- \( s \circ t \triangleq [i \in 1 .. (\text{Len}(s) + \text{Len}(t)) \mapsto \begin{cases} \text{if } i \leq \text{Len}(s) \text{ then } s[i] \text{ else } t[i - \text{Len}(s)] \end{cases}] \)

**THEOREM** ISpec \( \Rightarrow \square \text{TypeInvariant} \)**
This definition is illegal because the occurrence of \( \text{fact} \) to the right of the \( \triangleq \) is undefined—\( \text{fact} \) is defined only after its definition.

TLA\(^+\) does allow the apparent circularity of recursive function definitions. We can define the factorial function \( \text{fact} \) by:

\[
\text{fact}[n \in \text{Nat}] \triangleq \text{if } n = 0 \text{ then } 1 \text{ else } n \cdot \text{fact}[n - 1]
\]

In general, a definition of the form \( f[x \in S] \triangleq e \) can be used to define recursively a function \( f \) with domain \( S \). Section 6.3 explains exactly what such a definition means. For now, we will just write recursive definitions without worrying about their meaning.

### 5.6 A Write-Through Cache

We now specify a simple write-through cache that implements the memory specification. The system is described by the following picture:

Each processor \( p \) communicates with a local controller, which maintains three state components: \( \text{buf}[p] \), \( \text{ctl}[p] \), and \( \text{cache}[p] \). The value of \( \text{cache}[p] \) represents the processor’s cache; \( \text{buf}[p] \) and \( \text{ctl}[p] \) play the same role as in the internal memory specification (module \textit{InternalMemory}). (However, as we will see below, \( \text{ctl}[p] \) can assume an additional value “\text{waiting}”. ) These local controllers communicate with the main memory \( \text{mem} \)\(^4\) and with one another, over a bus. Requests from the processors to the main memory are in the queue \( \text{memQ} \) of maximum length \( QLen \).

\(^4\)This main memory does not directly correspond to the memory represented by variable \( \text{mem} \) in module \textit{InternalMemory}.\)
A write request by processor $p$ is performed by the action $DoWr(p)$. This is a write-through cache, meaning that every write request updates main memory. So, the $DoWr(p)$ action writes the value into $cache[p]$ and adds the write request to the tail of $memQ$. It also updates $cache[q]$ for any other processor $q$ that has a copy of the address in its cache. When the request reaches the head of $memQ$, the action $MemQWr$ stores the value in $mem$.

A read request by processor $p$ is performed by the action $DoRd(p)$, which obtains the value from the cache. If the value is not in the cache, the action $RdMiss(p)$ adds the request to the tail of $memQ$ and sets $ctl[p]$ to “waiting”. When the enqueued request reaches the head of $memQ$, the action $MemQRd$ reads the value and puts it in $cache[p]$, enabling the $DoRd(p)$ action.

We might expect the $MemQRd$ action to read the value from $mem$. However, this could cause an error if there is a write to that address enqueued in $memQ$ behind the read request. In that case, reading the value from memory could lead to two processors having different values for the address in their caches: the one that issued the read request, and the one that issued the write request that followed the read in $memQ$. So, the $MemQRd$ action must read the value from the last write to that address in $memQ$, if there is such a write; otherwise, it reads the value from $mem$.

Eviction of an address from processor $p$’s cache is represented by a separate $Evict(p)$ action. Since all cached values have been written to memory, eviction does nothing but remove the address from the cache. There is no reason to evict an address until the space is needed, so in an implementation, this action would be executed only when a request for an uncached address is received from $p$ and $p$’s cache is full. But that’s a performance optimization; it doesn’t affect the correctness of the algorithm, so it doesn’t appear in the specification. We allow a cached address to be evicted from $p$’s cache at any time—except if the address was just put there by a $MemQRd$ action for the current request. This is the case when $ctl[p]$ equals “waiting” and $buf[p].adr$ equals the cached address.

The actions $Req(p)$ and $Rsp(p)$, which represent processor $p$ issuing a request and the memory issuing a reply to $p$, are the same as the corresponding actions of the memory specification, except that they also leave the new variables $cache$ and $memQ$ unchanged.

To specify all these actions, we must decide how the processor caches and the queue of requests to memory are represented by the variables $memQ$ and $cache$. We let $memQ$ be a sequence of pairs of the form $(p, req)$, where $req$ is a request and $p$ is the processor that issued it. For any memory address $a$, we let $cache[p][a]$ be the value in $p$’s cache for address $a$ (the “copy” of $a$ in $p$’s cache). If $p$’s cache does not have a copy of $a$, we let $cache[p][a]$ equal $NoVal$.

The specification appears in module $WriteThroughCache$ on pages 56–58. I’ll now go through this specification, explaining some of the finer points and some notation that we haven’t encountered before.
 MODULE WriteThroughCache

EXTENDS Naturals, Sequences, MemoryInterface

VARIABLES mem, ctl, buf, cache, memQ

CONSTANT QLen

ASSUME (QLen \in Nat) \land (QLen > 0)

\( M \triangleq \text{instance InternalMemory} \)

\begin{align*}
\text{Init} & \triangleq \text{The initial predicate.} \\
\land M!\text{Init} & \land \text{mem, buf, and ctl are initialized as in the internal memory spec.} \\
\land \text{cache} & \land \text{All caches are initially empty (cache[p][a] = NoVal for all p, a).} \\
\land \text{memQ} = \{\} & \land \text{The queue memQ is initially empty.}
\end{align*}

\begin{align*}
\text{TypeInvariant} & \triangleq \text{The type invariant.} \\
\land \text{mem} \in [\text{Adr} \rightarrow \text{Val}] & \land \text{ctl} \in [\text{Proc} \rightarrow \{\text{"rdy"}, \text{"busy"}, \text{"waiting"}, \text{"done"}\}] \\
\land \text{buf} \in [\text{Proc} \rightarrow \text{MReq} \cup \text{Val} \cup \{\text{NoVal}\}] & \land \text{cache} \in [\text{Proc} \rightarrow [\text{Adr} \rightarrow \text{Val} \cup \{\text{NoVal}\}]] \\
\land \text{memQ} \in \text{Seq(Proc} \times \text{MReq}) & \land \text{memQ is a sequence of (proc., request) pairs.}
\end{align*}

\begin{align*}
\text{Coherence} & \triangleq \text{Asserts that if two processors' caches both have copies of an} \\
\land \forall p, q \in \text{Proc}, a \in \text{Adr} : & \land \text{address, then those copies have equal values.} \\
\land (\text{NoVal} \notin \{\text{cache}[p][a], \text{cache}[q][a]\}) & \land (\text{cache}[p][a] = \text{cache}[q][a])
\end{align*}

\begin{align*}
\text{Req}(p) & \triangleq \text{Processor p issues a request.} \\
M!\text{Req}(p) & \land \text{UNCHANGED (cache, memQ)}
\end{align*}

\begin{align*}
\text{Rsp}(p) & \triangleq \text{The system issues a response to processor p.} \\
M!\text{Rsp}(p) & \land \text{UNCHANGED (cache, memQ)}
\end{align*}

\begin{align*}
\text{RdMiss}(p) & \triangleq \text{Enqueue a request to write value from memory to p's cache.} \\
\land (\text{ctl}[p] = \text{"busy"}) & \land (\text{buf}[p].\text{op} = \text{"Rd"}) \text{ Enabled on a read request when} \\
\land \text{cache}[p][\text{buf}[p].\text{adr}] = \text{NoVal} & \land \text{the address is not in p's cache} \\
\land \text{Len}(\text{memQ}) < \text{QLen} & \land \text{and memQ is not full.} \\
\land \text{memQ}' = \text{Append}(\text{memQ}, (p, \text{buf}[p])) & \text{Append } <p, \text{request}> \text{ to memQ.} \\
\land \text{ctl}' = [\text{ctl} \text{ EXCEPT } ![p] = \text{"waiting"}] & \text{Set ctl}[p] \text{ to \"waiting\".} \\
\land \text{UNCHANGED } <\text{memInt, mem, buf, cache}> & \text{ }
\end{align*}

Figure 5.5: The write-through cache specification (beginning).
5.6. A WRITE-THROUGH CACHE

DoRd(p) \[=\] Perform a read by \( p \) of a value in its cache.
\( \land \ ctl[p] \in \{\text{“busy”}, \text{“waiting”}\} \)
\( \land \ buf[p].op = \text{“Rd”} \)
\( \land \ cache[p] = buf[p] \)
\( \land \ buf'[p] = [buf \ except ![p] = cache[p][buf[p].[adr]]] \)
\( \land \ ctl'[p] = [ctl \ except ![p] = \text{“done”}] \)
\( \land \ \text{UNCHANGED} \langle \text{memInt}, \text{mem}, \text{cache}, \text{memQ} \rangle \)

DoWr(p) \[=\] Write to \( p \)'s cache, update other caches, and enqueue memory update.

\[ \text{LET } r \triangleq buf[p] \quad \text{Processor } p\text{'s request.} \]
\( \land \ (ctl[p] = \text{“busy”}) \land (r.op = \text{“Wr”}) \quad \text{Enabled if write request pending} \)
\( \land \ \text{Len(memQ)} < QLen \quad \text{and memQ is not full.} \)
\( \land \ cache' = \text{Update } p\text{'s cache and any other cache that has a copy.} \)
\[ q \in \text{Proc} \rightarrow \text{IF } (p = q) \lor (cache[q][r.[adr]] \neq \text{NoVal}) \]
\[ \text{THEN } [cache[q] \ except ![r.[adr]] = r.val] \]
\[ \text{ELSE } cache[q] \]
\( \land \ memQ' = \text{Append(memQ, } (p, r)) \quad \text{Enqueue write at tail of memQ.} \)
\( \land \ buf'[p] = [buf \ except ![p] = \text{NoVal}] \quad \text{Generate response.} \)
\( \land \ ctl'[p] = [ctl \ except ![p] = \text{“done”}] \quad \text{Set ctl to indicate request is done.} \)
\( \land \ \text{UNCHANGED} \langle \text{memInt}, \text{mem} \rangle \)

vmem \[=\] The value \( \text{mem} \) will have after all the writes in \( \text{memQ} \) are performed.

\[ \text{LET } f[i \in 0 \ldots \text{Len(memQ)}] \triangleq \text{The value } \text{mem} \text{ will have after the first} \]
\[ i \text{ writes in } \text{memQ} \text{ are performed.} \]
\[ \text{IF } i = 0 \ \text{THEN mem} \]
\[ \text{ELSE IF memQ}[i][2].op = \text{“Rd”} \]
\[ \text{THEN } f[i-1] \]
\[ \text{ELSE } f[i-1] \ except ![memQ[i][2].[adr]] = \]
\[ \text{memQ[i][2].val} \]
\( \in \ f[\text{Len(memQ)}] \)

MemQWr \[=\] Perform write at head of \( \text{memQ} \) to memory.

\[ \text{LET } r \triangleq \text{Head(memQ)}[2] \quad \text{The request at the head of memQ.} \]
\( \land \ (memQ \neq \langle \rangle) \land (r.op = \text{“Wr”}) \quad \text{Enabled if Head(memQ) a write.} \)
\( \land \ mem' = \]
\[ \text{mem except ![r.[adr]] = r.val} \quad \text{Perform the write to memory.} \)
\( \land \ memQ' = \text{Tail(memQ)} \quad \text{Remove the write from memQ.} \)
\( \land \ \text{UNCHANGED} \langle \text{memInt}, \text{mem}, 	ext{buf}, \text{ctl}, \text{cache} \rangle \)

Figure 5.6: The write-through cache specification (middle).
MEMQRD \(=\) Perform an enqueued read to memory.

\[
\begin{align*}
\text{LET } p & \triangleq \text{Head}(\text{memQ})[1] & \text{The requesting processor.} \\
r & \triangleq \text{Head}(\text{memQ})[2] & \text{The request at the head of memQ.} \\
n & \triangleq (\text{memQ} \neq \langle \rangle) \land (r.op = \text{"Rd"}) & \text{Enabled if Head(memQ) is a read.} \\
\land \, \text{memQ}^' & \triangleq \text{Tail}(\text{memQ}) & \text{Remove the head of memQ.} \\
\land \, \text{cache}' & = & \text{Put value from memory or memQ in } p\text{’s cache.}
\end{align*}
\]

\[
\begin{align*}
\text{[cache EXCEPT ![p][r.adr] = vmem[r.adr]]} \\
\land \, \text{UNCHANGED } & \langle \text{memInt, mem, buf, ctl} \rangle
\end{align*}
\]

\[
\begin{align*}
\text{Evict}(p, a) & \triangleq \text{Remove address } a \text{ from } p\text{’s cache.} \\
\land \, (\text{ctl}[p] = \text{“waiting”}) & \Rightarrow (\text{buf}[p].adr \neq a) & \text{Can’t evict } a \text{ if it was just read into cache from memory.} \\
\land \, \text{cache}' & = [\text{cache EXCEPT ![p][a] = NoVal}] \\
\land \, \text{UNCHANGED } & \langle \text{memInt, mem, buf, ctl, memQ} \rangle
\end{align*}
\]

\[
\begin{align*}
\text{Next} & \triangleq \vee \exists p \in \text{Proc} : \vee \text{Req}(p) \lor \text{Rsp}(p) \\
\land \, \vee \text{RdMiss}(p) \lor \text{DoRd}(p) \lor \text{DoWr}(p) \\
\land \, \exists a \in \text{Addr} : \text{Evict}(p, a) \\
\land \, \text{MemQWr} \lor \text{MemQRd}
\end{align*}
\]

\[
\begin{align*}
\text{Spec} & \triangleq \text{Init} \land \Box[\text{Next}]_{\langle \text{memInt, mem, buf, ctl, memQ} \rangle}
\end{align*}
\]

**THEOREM** Spec \(\Rightarrow\) \(\Box(\text{TypeInvariant} \land \text{Coherence})\)

\[
\begin{align*}
\text{LM} & \triangleq \text{INSTANCE Memory} & \text{The memory spec. with internal variables hidden.} \\
\text{THEOREM} \, \text{Spec} & \Rightarrow \text{LM} \land \text{Spec} & \text{Formula Spec implements the memory spec.}
\end{align*}
\]

Figure 5.7: The write-through cache specification (end).

The extends, declaration statements, and assume are familiar. We can re-use some of the definitions from the InternalMemory module, so an instance statement instantiates a copy of that module. (The parameters of module InternalMemory are instantiated by the parameters of the same name in module WriteThroughCache.)

The initial predicate Init contains the conjunct \(M!\text{Init}\), which asserts that mem, ctl, and buf have the same initial values as in the internal memory specification. The write-through cache allows ctl\([p]\) to have the value “waiting” that it didn’t in the internal memory specification, so we can’t re-use the internal memory’s type invariant \(M!\text{TypeInvariant}\). Formula TypeInvariant therefore explicitly describes the types of mem, ctl, and buf. The type of memQ is the set of sequences of \((\text{processor, request})\) pairs.

The module next defines the predicate Coherence, which asserts the basic cache coherence property of the write-through cache: for any processors \(p\) and \(q\)
and any address \(a\), if \(p\) and \(q\) each has a copy of address \(a\) in its cache, then those copies are equal. Note the trick of writing \(x \notin \{y, z\}\) instead of the equivalent but longer formula \((x \neq y) \land (x \neq z)\).

The actions \(\text{Req}(p)\) and \(\text{Rsp}(p)\), which represent a processor sending a request and receiving a reply, are essentially the same as the corresponding actions in module \(\text{InternalMemory}\). The only difference is that they must specify that the variables \(\text{cache}\) and \(\text{memQ}\), not present in module \(\text{InternalMemory}\), are left unchanged.

In the definition of \(\text{RdMiss}\), the expression \(\text{Append}(\text{memQ}, \langle p, \text{buf}[p]\rangle)\) is the sequence obtained by appending the element \(\langle p, \text{buf}[p]\rangle\) to the end of \(\text{memQ}\).

The \(\text{DoRd}(p)\) action represents the performing of the read from \(p\)'s cache. If \(\text{ctl}[p] = \text{“busy”}\), then the address was originally in the cache. If \(\text{ctl}[p] = \text{“waiting”}\), then the address was just read into the cache from memory.

The \(\text{DoWr}(p)\) action writes the value to \(p\)'s cache and updates the value in any other caches that have copies. It also enqueues a write request in \(\text{memQ}\). In an implementation, the request is put on the bus, which transmits it to the other caches and to the \(\text{memQ}\) queue. In our high-level view of the system, we represent all this as a single step.

The definition of \(\text{DoWr}\) introduces the TLA\(^+\) \texttt{let/in} construct. The \texttt{let} clause consists of a sequence of definitions, whose scope extends until the end of the \texttt{in} clause. In the definition of \(\text{DoWr}\), the \texttt{let} clause defines \(r\) to equal \(\text{buf}[p]\) within the \texttt{in} clause. Observe that the definition of \(r\) contains the parameter \(p\) of the definition of \(\text{DoWr}\). Hence, we could not move the definition of \(r\) outside the definition of \(\text{DoWr}\).

A \texttt{let} can also be used to make an expression easier to read, even if the operators it defines appear only once in the \texttt{in} expression. We write a specification with a sequence of definitions, instead of just defining a single monolithic formula, because a formula is easier to understand when presented in smaller chunks. The \texttt{let} construct allows the process of splitting a formula into smaller parts to be done hierarchically. A \texttt{let} can appear as a subexpression of an \texttt{in} expression. Nested \texttt{lets} are common in large, complicated specifications.

Next comes the definition of the state function \(\text{vmem}\), which is used in defining action \(\text{MemQRd}\) below. It equals the value that the main memory \(\text{mem}\) will have after all the write operations currently in \(\text{memQ}\) have been performed. Recall that the value read by \(\text{MemQRd}\) must be the most recent one written
to that address—a value that may still be in \textit{memQ}. That value is the one in \textit{vmem}. The function \textit{vmem} is defined in terms of the recursively defined function \textit{f}, where \(f[i]\) is the value \textit{mem} will have after the first \(i\) operations in \textit{memQ} have been performed. Note that \(\textit{memQ}[i][2]\) is the second component (the request) of \(\textit{memQ}[i]\), the \(i\)th element in the sequence \textit{memQ}.

The next two actions, \textit{MemQWr} and \textit{MemQRd}, represent the processing of the request at the head of the \textit{memQ} queue—\textit{MemQWr} for a write request, and \textit{MemQRd} for a read request. These actions also use a \texttt{let} to make local definitions. Here, the definitions of \(p\) and \(r\) could be moved before the definition of \textit{MemQWr}. In fact, we could save space by replacing the two local definitions of \(r\) with one global (within the module) definition. However, making the definition of \(r\) global in this way would be somewhat distracting, since \(r\) is used only in the definitions of \textit{MemQWr} and \textit{MemQRd}. It might be better instead to combine these two actions into one. Whether you put a definition into a \texttt{let} or make it more global should depend on what makes the specification easier to read.

Writing specifications is a craft whose mastery requires talent and hard work. The \texttt{Evict}(\(p, a\)) action represents the operation of removing address \(a\) from processor \(p\)'s cache. As explained above, we allow an address to be evicted at any time—unless the address was just written to satisfy a pending read request, which is the case iff \(ctl[p] = \text{"waiting"}\) and \(buf[p].adr = a\). Note the use of the “double subscript” in the \texttt{EXCEPT} expression of the action’s second conjunct. This conjunct “assigns \textit{NoVal} to cache\([p][a]\)”. If address \(a\) is not in \(p\)'s cache, then \(cache[p][a]\) already equals \textit{NoVal} and an \texttt{Evict}(\(p, a\)) step is a stuttering step.

The definitions of the next-state action \texttt{Next} and of the complete specification \textit{Spec} are straightforward. The module closes with two theorems that are discussed below.

### 5.7 Invariance

Module \textit{WriteThroughCache} contains the theorem

\[
\text{THEOREM } \textit{Spec} \Rightarrow \Box (\text{TypeInvariant} \land \text{Coherence})
\]

which asserts that \textit{TypeInvariant} \land \textit{Coherence} is an invariant of \textit{Spec}. A state predicate \(P \land Q\) is always true iff both \(P\) and \(Q\) are always true, so \(\Box(P \land Q)\) is equivalent to \(\Box P \land \Box Q\). This implies that the theorem above is equivalent to the two theorems:

\[
\text{THEOREM } \textit{Spec} \Rightarrow \Box \text{TypeInvariant}
\]

\[
\text{THEOREM } \textit{Spec} \Rightarrow \Box \text{Coherence}
\]

The first theorem is the usual type-invariance assertion. The second, which asserts that \textit{Coherence} is an invariant of \textit{Spec}, expresses an important property of the algorithm.
Although \textit{TypeInvariant} and \textit{Coherence} are both invariants of the temporal formula \textit{Spec}, they differ in a fundamental way. If \( s \) is any state satisfying \textit{TypeInvariant}, then any state \( t \) such that \( s \rightarrow t \) is a \textit{Next} step also satisfies \textit{TypeInvariant}. This property is expressed by:

\textbf{Theorem} \( \textit{TypeInvariant} \land \textit{Next} \Rightarrow \textit{TypeInvariant}' \)

(Recall that \textit{TypeInvariant}' is the formula obtained by priming all the variables in formula \textit{TypeInvariant}.) In general, when \( P \land N \Rightarrow P' \) holds, we say that predicate \( P \) is an invariant of action \( N \). Predicate \textit{TypeInvariant} is an invariant of \textit{Spec} because it is an invariant of \textit{Next} and it is implied by the initial predicate \textit{Init}.

Predicate \textit{Coherence} is not an invariant of the next-state action \textit{Next}. Suppose \( s \) is a state in which

- \texttt{cache}[p1][a] = 1
- \texttt{cache}[q][b] = \texttt{NoVal}, for all \( \langle q, b \rangle \) different from \( \langle p1, a \rangle \)
- \texttt{mem}[a] = 2
- \texttt{memQ} contains the single element \( \langle p2, \langle op \leftrightarrow \texttt{Rd}, adr \leftrightarrow a \rangle \} \)

for two different processors \( p1 \) and \( p2 \) and some address \( a \). Then \textit{Coherence} is true in state \( s \). Let \( t \) be the state obtained from \( s \) by taking a \texttt{MemQRd} step. In state \( t \), we have \texttt{cache}[p2][a] = 2 and \texttt{cache}[p1][a] = 1, so \textit{Coherence} is false. Hence \textit{Coherence} is not an invariant of the next-state action.

\textit{Coherence} is an invariant of formula \textit{Spec} because states like \( s \) cannot occur in a behavior satisfying \textit{Spec}. Proving its invariance is not so easy. We must find a predicate \( \textit{Inv} \) that is an invariant of \textit{Next} such that \( \textit{Inv} \) implies \textit{Coherence} and is implied by the initial predicate \textit{Init}.

Important properties of a specification can often be expressed as invariants. Proving that a state predicate \( P \) is an invariant of a specification means proving a formula of the form

\[ \textit{Init} \land \Box [\textit{Next}]_v \Rightarrow \Box P \]

This is done by finding an appropriate state predicate \( \textit{Inv} \) and proving

\[ \textit{Init} \Rightarrow \textit{Inv}, \quad \textit{Inv} \land [\textit{Next}]_v \Rightarrow \textit{Inv}', \quad \textit{Inv} \Rightarrow P \]

Since our subject is specification, not proof, I won’t discuss how to find \( \textit{Inv} \).

\section{5.8 Proving Implementation}

Module \textit{WriteThroughCache} ends with the theorem

\textbf{Theorem} \textit{Spec} \Rightarrow \textit{LM1Spec}
where $LM!Spec$ is formula $Spec$ of module $Memory$. By definition of this formula (page 53), we can restate the theorem as

$$\text{THEOREM } Spec \Rightarrow \exists \text{ mem, ctl, buf : } LM!Inner(\text{mem, ctl, buf})!ISpec$$

where $LM!Inner(\text{mem, ctl, buf})!ISpec$ is formula $ISpec$ of the $InnerMemory$ module. The rules of logic tell us that to prove such a theorem, we must find “witnesses” for the quantified variables $\text{mem}$, $\text{ctl}$, and $\text{buf}$. These witness are state functions (ordinary expressions with no primes), which I’ll call $\text{omem}$, $\text{octl}$, and $\text{obuf}$, that satisfy:

$$(5.1) \quad Spec \Rightarrow LM!Inner(\text{omem, octl, ubuf})!ISpec$$

The tuple $(\text{omem, octl, ubuf})$ of witness functions is called a refinement mapping, and we describe (5.1) as the assertion that $Spec$ implements formula $ISpec$ (of module $InnerMemory$) under this refinement mapping. Intuitively, this means $Spec$ implies that the value of the tuple of state functions $(\text{memInt, mem, ctl, buf})$ changes the way $ISpec$ asserts that the tuple of variables $(\text{memInt, mem, ctl, buf})$ should change.

I will now briefly describe how we prove (5.1); for details, see [3]. Let me first introduce a bit of non-TLA$^+$ notation. For any formula $F$ of module $InnerMemory$, let $\overline{F}$ equal $LM!Inner(\text{omem, octl, ubuf})!F$, which is formula $F$ with $\text{omem}$, $\text{octl}$, and $\text{obuf}$ substituted for $\text{mem}$, $\text{ctl}$, and $\text{buf}$. In particular, $\overline{\text{mem}}$, $\overline{\text{ctl}}$, and $\overline{\text{buf}}$ equal $\text{omem}$, $\text{octl}$, and $\text{obuf}$, respectively.

Replacing $Spec$ and $ISpec$ by their definitions transforms (5.1) to

$$\text{Init} \land \Box[Next](\text{memInt, mem, buf, ctl, cache, memQ})$$

$$\Rightarrow \overline{\text{Init}} \land \Box[Next](\text{memInt, mem, ctl, ubuf})$$

This is proved by finding an invariant $Inv$ of $Spec$ such that

$$\land \text{ Init } \Rightarrow \overline{\text{Init}}$$

$$\land \text{ Inv } \land \text{ Next } \Rightarrow \lor \overline{\text{Next}}$$

$$\lor \text{ UNCHANGED } (\text{memInt, mem, ctl, ubuf})$$

The second conjunct is called step simulation. It asserts that a $\text{Next}$ step starting in a state satisfying the invariant $\text{Inv}$ is either an $\overline{\text{Next}}$ step—a step that changes the 4-tuple $(\text{memInt, omem, octl, ubuf})$ the way an $\overline{\text{Next}}$ step changes $(\text{memInt, mem, ctl, buf})$—or else it leaves that 4-tuple unchanged.

The mathematics of an implementation proof is simple, so the proof is straightforward—in theory. For specifications of real systems, such proofs can be quite difficult. Going from the theory to practice requires turning the mathematics of proofs into an engineering discipline—a subject that deserves a book to itself. However, when writing specifications, it helps to understand refinement mappings and step simulation.
5.8. PROVING IMPLEMENTATION

We now return to the question posed in Section 3.2: what is the relation between the specifications of the asynchronous interface in modules AsynchInterface and Channel? Recall that module AsynchInterface describes the interface in terms of the three variables \( \text{val}, \text{rdy}, \text{and} \ \text{ack} \), while module Channel describes it with a single variable \( \text{chan} \) whose value is a record with \( \text{val}, \text{rdy}, \text{and} \ \text{ack} \) components. In what sense are those two specifications of the interface equivalent?

One answer that now suggests itself is that each of the specifications should implement the other under a refinement mapping. We expect formula \( \text{Spec} \) of module Channel to imply the formula obtained from \( \text{Spec} \) of module AsynchInterface by substituting for its variables \( \text{val}, \text{rdy}, \text{and} \ \text{ack} \) the \( \text{val}, \text{rdy}, \text{and} \ \text{ack} \) components of the variable \( \text{chan} \) of module Channel. This assertion is expressed precisely by the theorem in the following module.

```
MODULE ChannelImplAsynch

EXTENDS Channel
AInt(val; rdy; ack) ≜ INSTANCE AsynchInterface
THEOREM Spec ⇒ AInt(chan.val; chan.rdy; chan.ack)!Spec
```

Here, the refinement mapping substitutes \( \langle \text{chan.val}, \text{chan.rdy}, \text{chan.ack} \rangle \) for the tuple \( \langle \text{val}, \text{rdy}, \text{ack} \rangle \) of variables in the formula \( \text{Spec} \) of module AsynchInterface.

Similarly, formula \( \text{Spec} \) of module AsynchInterface implies formula \( \text{Spec} \) of module Channel with \( \text{chan} \) replaced by the record-valued expression:

\[
\left[ \text{val} \mapsto \text{val}, \ \text{rdy} \mapsto \text{rdy}, \ \text{ack} \mapsto \text{ack} \right]
\]

(The first \( \text{val} \) in \( \text{val} \mapsto \text{val} \) is the field name in the record constructor, while the second \( \text{val} \) is the variable of module AsynchInterface.)
Chapter 6

Some More Math

Our mathematics is built on a small, simple collection of concepts. You’ve already seen most of what’s needed to describe almost any kind of mathematics. All you lack are a handful of operators on sets that are described below in Section 6.1. After learning about them, you will be able to define all the data structures and operations that occur in specifications.

While our mathematics is simple, its foundations are nonobvious—for example, the meanings of recursive function definitions and the \textsc{choose} operator are subtle. This section discusses some of those foundations. Understanding them will help you use mathematics more effectively.

6.1 Sets

The simple operations on sets described in Section 1.2 are all you’ll need for writing most system specifications. However, you may occasionally have to use more sophisticated operators—especially if you need to define data structures beyond tuples, records, and simple functions.

Two powerful operators of set theory are the unary operators \textsc{union} and \textsc{subset}, defined as follows.

**UNION** $S$ The union of the elements of $S$. In other words, a value $e$ is an element of \textsc{union} $S$ iff it is an element of an element of $S$. For example:

\[
\text{UNION} \ \{\{1,2\}, \{2,3\}, \{3,4\}\} = \{1,2,3,4\}
\]

**SUBSET** $S$ The set of all subsets of $S$. In other words, $T \in \textsc{subset} S$ iff $T \subseteq S$. For example:

\[
\text{SUBSET} \ \{1,2\} = \{\{\}, \{1\}, \{2\}, \{1,2\}\}
\]

Mathematicians write \textsc{union} $S$ as $\bigcup S$.

Mathematicians call \textsc{union} $S$ the powerset of $S$ and write it $P(S)$ or $2^S$. 
Mathematicians often describe a set as “the set of all . . . such that . . .”. TLA+ has two constructs that formalize such a description:

\[ \{ x \in S : P \} \]  
The subset of \( S \) consisting of all elements \( x \) satisfying property \( P \). For example, the set of odd natural numbers can be written \( \{ n \in \text{Nat} : n \% 2 = 1 \} \). The identifier \( x \) is bound in \( P \); it may not occur in \( S \).

\[ \{ e : x \in S \} \]  
The set of elements of the form \( e \), for all \( x \) in the set \( S \). For example, \( \{2 * n + 1 : n \in \text{Nat} \} \) is the set of all odd natural numbers. The identifier \( x \) is bound in \( e \); it may not occur in \( S \).

The construct \( \{ e : x \in S \} \) has the same generalizations as \( \exists x \in S : F \). For example, \( \{ e : x \in S, y \in T \} \) is the set of all elements of the form \( e \), for \( x \) in \( S \) and \( y \) in \( T \). In the construct \( \{ x \in S : P \} \), we can let \( x \) be a tuple. For example, \( \{(y, z) \in S : P \} \) is the set of all pairs \((y, z)\) in the set \( S \) that satisfy \( P \). The BNF grammar of TLA+ in Section 14.1.2 specifies precisely what set expressions you can write.

All the set operators we’ve seen so far are built-in operators of TLA+. There is also a standard module \textit{FiniteSets} that defines two operators:

\[ \text{Cardinality}(S) \]  
The number of elements in set \( S \), if \( S \) is a finite set.

\[ \text{IsFiniteSet}(S) \]  
True iff \( S \) is a finite set.

Careless reasoning about sets can lead to problems. The classic example of this is Russell’s paradox:

Let \( \mathcal{R} \) be the set of all sets \( S \) such that \( S \notin S \). The definition of \( \mathcal{R} \) implies that \( \mathcal{R} \) is an element of \( \mathcal{R} \) iff \( \mathcal{R} \) is not an element of \( \mathcal{R} \) is true.

Obviously, \( \mathcal{R} \) can’t both be and not be an element of \( \mathcal{R} \). The source of the paradox is that \( \mathcal{R} \) isn’t a set. There’s no way to write it in TLA+. Intuitively, \( \mathcal{R} \) is too big to be a set. A collection \( \mathcal{C} \) is too big to be a set if it is as big as the collection of all sets—meaning that we can assign to every set a different element of \( \mathcal{C} \). More precisely, \( \mathcal{C} \) is too big to be a set if we can define an operator \textit{SMap} such that:

- \( SMap(S) \) is in \( \mathcal{C} \), for any set \( S \).
- If \( S \) and \( T \) are two different sets, then \( SMap(S) \neq SMap(T) \).

For example, the collection of all sequences of length 2 is too big to be a set; we can define the operator \textit{SMap} by

\[ SMap(S) \triangleq (1, S) \]
6.2 Silly Expressions

Most modern programming languages introduce some form of type checking to prevent you from writing silly expressions like $3/"abc"$. TLA+ is based on the usual formalization of mathematics, which doesn’t have types. In an untyped formalism, every syntactically well-formed expression has a meaning—even a silly expression like $3/"abc"$. Mathematically, the expression $3/"abc"$ is no sillier than the expression $3/0$, and mathematicians implicitly write that silly expression all the time. For example, consider the valid formula

$$\forall x \in \text{Real} : (x \neq 0) \Rightarrow (x * (3/x) = 3)$$

where Real is the set of all real numbers. This asserts that $(x \neq 0) \Rightarrow (x * (3/x) = 3)$ is true for all real numbers $x$. Substituting 0 for $x$ yields the valid formula $(0 \neq 0) \Rightarrow (0 * (3/0) = 3)$ that contains the silly expression $3/0$. It’s valid because $0 \neq 0$ equals FALSE, and FALSE $\Rightarrow P$ is true for any formula $P$.

A correct formula can contain silly expressions. For example, $3/0 = 3/0$ is a correct formula because any value equals itself. However, the validity of a correct formula cannot depend on the meaning of a silly expression. If an expression is silly, then its meaning is probably unspecified. The definitions of 0, 3, $\times$, and $/$ (which are in the standard module Reals) don’t specify the value of $0 * (3/0)$, so there’s no way of knowing whether that value equals 3.

No sensible syntactic rules can prevent you from writing $3/0$ without also preventing you from writing perfectly reasonable expressions. The typing rules of programming languages introduce complexity and limitations on what you can write that don’t exist in ordinary mathematics. In a well-designed programming language, the costs of types are balanced by benefits: types allow a compiler to produce more efficient code, and type checking catches errors. For programming languages, the benefits seem to outweigh the costs. For writing specifications, I have found that the costs outweigh the benefits.

If you’re used to the constraints of programming languages, it may be a while before you start taking advantage of the freedom afforded by mathematics. At first, you won’t think of defining anything like the operator $R$ defined on page 50 of Section 5.2, which couldn’t be written in a typed programming language.

6.3 Recursive Function Definitions Revisited

Section 5.5 introduced recursive function definitions. Let’s now examine what such definitions mean mathematically. Mathematicians usually define the factorial function $fact$ by writing:

$$fact[n] = \text{IF } n = 0 \text{ THEN } 1 \text{ ELSE } n \times fact[n - 1], \text{ for all } n \in \text{Nat}$$
This definition can be justified by proving that it defines a unique function \( \text{fact} \) with domain \( \text{Nat} \). In other words, \( \text{fact} \) is the unique value satisfying:

\[
(6.1) \quad \text{fact} = [n \in \text{Nat} \mapsto \text{if } n = 0 \text{ then } 1 \text{ else } n * \text{fact}[n - 1]]
\]

The \text{choose} operator, introduced on pages 47–48 of Section 5.1, allows us to express “the value satisfying property \( P \)” as \text{choose} \( x : P \). We can therefore define \( \text{fact} \) as follows to be the value satisfying (6.1):

\[
(6.2) \quad \text{fact} \overset{\Delta}{=} \text{choose} \text{fact} : \\
\text{fact} = [n \in \text{Nat} \mapsto \text{if } n = 0 \text{ then } 1 \text{ else } n * \text{fact}[n - 1]]
\]

(Since the symbol \( \text{fact} \) is not yet defined in the expression to the right of the \( \overset{\Delta}{=} \), we can use it as the bound identifier in the \text{choose} expression.) The TLA\(^+\) definition

\[
\text{fact}[n \in \text{Nat}] \overset{\Delta}{=} \text{if } n = 0 \text{ then } 1 \text{ else } n * \text{fact}[n - 1]
\]

is simply an abbreviation for (6.2). In general, \( f[x \in S] \overset{\Delta}{=} e \) is an abbreviation for:

\[
(6.3) \quad f \overset{\Delta}{=} \text{choose } f : f = [x \in S \mapsto e]
\]

TLA\(^+\) allows you to write silly definitions. For example, you can write

\[
(6.4) \quad \text{circ}[n \in \text{Nat}] \overset{\Delta}{=} \text{choose } y : y \neq \text{circ}[n]
\]

This appears to define \( \text{circ} \) to be a function such that \( \text{circ}[n] \neq \text{circ}[n] \) for any natural number \( n \). There obviously is no such function, so \( \text{circ} \) can’t be defined to equal it. A recursive function definition doesn’t necessarily define a function. If there is no \( f \) that equals \( [x \in S \mapsto e] \), then (6.3) defines \( f \) to be some unspecified value. Thus, the nonsensical definition (6.4) defines \( \text{circ} \) to be some unknown value.

If we want to reason about a function \( f \) defined by \( f[x \in S] \overset{\Delta}{=} e \), we need to prove that there exists an \( f \) that equals \( [x \in S \mapsto e] \). The existence of \( f \) is obvious if \( f \) does not occur in \( e \). If it does, so this is a recursive definition, then there is something to prove. Since I’m not discussing proofs, I won’t describe how to prove it. Intuitively, you have to check that, as in the case of the factorial function, the definition uniquely determines the value of \( f[x] \) for every \( x \) in \( S \).

Recursion is a common programming technique because programs must compute values using a small repertoire of simple elementary operations. It’s not used as often in mathematical definitions, where we needn’t worry about how to compute the value and can use the powerful operators of logic and set theory. For example, the operators \text{Head}, \text{Tail}, \text{and } \circ \text{ are defined in Section 5.4 without recursion, even though computer scientists usually define them recursively. Still, there are some things that are best defined inductively, using a recursive function definition.
6.4 Functions versus Operators

Consider these definitions, which we’ve seen before

\[
\text{Tail}(s) \triangleq [i \in 1 \ldots (\text{Len}(s) - 1) \mapsto s[i + 1]]
\]

\[
\text{fact}[n \in \text{Nat}] \triangleq \text{if } n = 0 \text{ then } 1 \text{ else } n \times \text{fact}[n - 1]
\]

They define two very different kinds of objects: \text{fact} is a function, and \text{Tail} is an operator. Functions and operators differ in a few basic ways.

Their most obvious difference is that a function like \text{fact} by itself is a complete expression that denotes a value, but an operator like \text{Tail} is not. Both \text{fact}[n] \in S and \text{fact} \in S are syntactically correct expressions. But, while \text{Tail}(n) \in S is syntactically correct, \text{Tail} \in S is not. It is gibberish—a meaningless string of symbols, like \(x+ = 0\).

Their second difference is more profound. The definition of \text{Tail} defines \text{Tail}(s) for all values of \(s\). For example, it defines \text{Tail}(1/2) to equal

\[
[i \in 1 \ldots (\text{Len}(1/2) - 1) \mapsto (1/2)[i + 1]]
\]

We have no idea what this expression means, because we don’t know what \(\text{Len}(1/2)\) or \((1/2)[i + 1]\) mean. But, whatever (6.5) means, it equals \text{Tail}(1/2).

The definition of \text{fact} defines \text{fact}[n] only for \(n \in \text{Nat}\). It tells us nothing about the value of \text{fact}[1/2]. The expression \text{fact}[1/2] is syntactically well-formed, so it denotes some value. But the definition of \text{fact} tells us nothing about what that value is.

Unlike an operator, a function must have a domain, which is a set. We cannot define a function \text{Tail} so that \text{Tail}[s] is the tail of any nonempty sequence \(s\); the domain of such a function would have to include all nonempty sequences, and the collection of all such sequences is too big to be a set. (The operator \text{SMap} defined by \text{SMap}(S) \triangleq \{S\} maps every set to a different nonempty sequence.) Hence, we can’t define \text{Tail} to be a function.

Unlike a function, an operator cannot be defined recursively. However, we can usually transform an illegal recursive operator definition into a nonrecursive one using a recursive function definition. For example, let’s try to define the \text{Cardinality} operator on the set of finite sets. (Recall that the cardinality of a finite set \(S\) is the number of elements in \(S\).) The collection of all finite sets is too big to be a set. (The operator \text{SMap}(S) \triangleq \{S\} maps every set \(S\) to a different set \(\{S\}\) of cardinality 1.) The \text{Cardinality} operator has a simple intuitive definition:

- \(\text{Cardinality}() = 0\).
- If \(S\) is a nonempty finite set, then

\[
\text{Cardinality}(S) = 1 + \text{Cardinality}(S \setminus \{x\})
\]

where \(x\) is an arbitrary element of \(S\). (The set \(S \setminus \{x\}\) contains all the elements of \(S\) except \(x\).)
Using the \texttt{choose} operator to describe an arbitrary element of \(S\), we can write this as the more formal-looking, but still illegal, definition:

\[
\text{Cardinality}(S) \triangleq \\
\text{if } S = \{\} \text{ then } 0 \\
\text{else } 1 + \text{Cardinality}(S \setminus \{\text{choose } x : x \in S\})
\]

This definition is illegal because it’s circular—only in a recursive function definition can the symbol being defined appear to the right of the \(\triangleq\).

To turn this into a legal definition, observe that, for a given set \(S\), we can define a function \(CS\) such that \(CS[T]\) equals the cardinality of \(T\) for every subset \(T\) of \(S\). The definition is

\[
CS[T \in \text{subset } S] \triangleq \\
\text{if } T = \{\} \text{ then } 0 \\
\text{else } 1 + CS[T \setminus \{\text{choose } x : x \in T\}]
\]

Since \(S\) is a subset of itself, this defines \(CS[S]\) to equal \(\text{Cardinality}(S)\), if \(S\) is a finite set. (We don’t know or care what \(CS[S]\) equals if \(S\) is not finite.) So, we can define the \texttt{Cardinality} operator by:

\[
\text{Cardinality}(S) \triangleq \\
\text{let } CS[T \in \text{subset } S] \triangleq \\
\text{if } T = \{\} \text{ then } 0 \\
\text{else } 1 + CS[T \setminus \{\text{choose } x : x \in T\}] \\
\text{in } CS[S]
\]

Operators also differ from functions in that an operator can take an operator as an argument. For example, we can define an operator \texttt{IsPartialOrder} so that \(\text{IsPartialOrder}(R, S)\) equals true iff the operator \(R\) defines an irreflexive partial order on \(S\). The definition is

\[
\text{IsPartialOrder}(R(x, y), S) \triangleq \\
\forall x, y, z \in S : R(x, y) \land R(y, z) \Rightarrow R(x, z) \\
\land \forall x \in S : \neg R(x, x)
\]

We could also use an infix-operator symbol like \(<\) instead of \(R\) as the parameter of the definition, writing:

\[
\text{IsPartialOrder}(x < y, S) \triangleq \\
\forall x, y, z \in S : (x < y) \land (y < z) \Rightarrow (x < z) \\
\land \forall x \in S : \neg (x < x)
\]

The first argument of \texttt{IsPartialOrder} is an operator that takes two arguments; its second argument is an expression. Since \(>\) is an operator that takes two arguments, the expression \texttt{IsPartialOrder}(>, \texttt{Nat}) is syntactically correct. In fact, it’s valid—if \(>\) is defined to be the usual operator on numbers. The expression
6.5. USING FUNCTIONS

IsPartialOrder(+, 3) is also syntactically correct, but it’s silly and we have no idea whether or not it’s valid.

The last difference between operators and functions has nothing to do with mathematics and is an idiosyncrasy of TLA+: the language doesn’t permit you to define infix functions. So, if we want to define /, we have no choice but to make it an operator.

One can write equally nonsensical things using functions or operators. However, whether you use functions or operators may determine whether the nonsense you write is nonsyntactic gibberish or syntactically correct but semantically silly. The string of symbols 2(“a”) is not a syntactically correct formula because 2 is not an operator. However, 2[“a”], which can also be written 2.a, is a syntactic correct expression. It’s nonsensical because 2 isn’t a function, so we don’t know what 2[“a”] means. Similarly, Tail(s, t) is syntactically incorrect because Tail is an operator that takes a single argument. However, as explained in Section 15.1.5 (page 192), fact[m, n] is syntactic sugar for fact[(m, n)], so it is a syntactically correct, semantically silly formula. Whether an error is syntactic or semantic determines what kind of tool can catch it. In particular, the parser described in Chapter 12 catches syntactic errors, but not semantic silliness.

The distinction between functions and operators seems to confuse some people. One reason is that, although this distinction exists in ordinary math, it usually goes unnoticed by mathematicians. If you point out to them that subset can’t be a function because its domain couldn’t be a set, mathematicians will realize that they use operators like subset and ∈ all the time. But, they never think of them as forming a distinct kind of entity. Logicians will observe that the distinction between operators and values, including functions, arises because TLA+ is a first-order logic rather than a higher-order logic.

When defining an object V, you may have to decide whether to make V an operator or a function. The differences between operators and functions will often determine the decision. For example, if a variable may have V as its value, then V must be a function. Thus, in the memory specification of Section 5.3, we had to represent the state of the memory by a function rather than an operator, since the variable mem couldn’t equal an operator. If these differences don’t determine whether to use an operator or a function, then it’s a matter of taste. I usually prefer operators.

6.5 Using Functions

Consider the following two formulas:

(6.6) \( f' = [i \in \text{Nat} \mapsto i + 1] \)

(6.7) \( \forall i \in \text{Nat} : f'[i] = i + 1 \)
These formulas both imply that $f'[i] = i + 1$ for every natural number $i$, but they are not equivalent. Formula (6.6) uniquely determines $f'$, asserting that it’s a function with domain $Nat$. Formula (6.7) is satisfied by lots of different values of $f'$—for example, by the function

$$ [i \in Real \rightarrow \text{if } i \in Nat \text{ then } i + 1 \text{ else } \sqrt{i} ] $$

In fact, from (6.7), we can’t even deduce that $f'$ is a function. Formula (6.6) implies formula (6.7), but not vice-versa.

When writing specifications, we almost always want to specify the new value of a variable $f$ rather than the new values of $f[i]$ for all $i$ in some set. We therefore usually write (6.6) rather than (6.7),

### 6.6 Choose

The choose operator was introduced in the memory interface of Section 5.1 in the simple idiom $\text{choose } v : v \notin S$, which is an expression whose value is not an element of $S$. In Section 6.3 above, we saw that it is a powerful tool that can be used in rather subtle ways.

The most common use for the choose operator is to "name" a uniquely specified value. For example, $a/b$ is the unique real number that satisfies the formula $a = b \times (a/b)$, if $a$ and $b$ are real numbers and $b \neq 0$. So, the standard module Reals defines division on the set $Real$ of real numbers by $a/b \overset{\Delta}{=} \text{choose } c \in Real : a = b \times c$

(The expression $\text{choose } x \in S : P$ means $\text{choose } x : (x \in S) \wedge P$.) If $a$ is a nonzero real number, then there is no real number $c$ such that $a = 0 \times c$. Therefore, $a/0$ has an unspecifed value. We don’t know what a real number times a string equals, so we cannot say whether or not there is a real number $c$ such that $a$ equals "xyz". Hence, we don’t know what the value of $a/"xyz"$ is.

People who do a lot of programming and not much mathematics often think that choose must be a nondeterministic operator. In mathematics, there is no such thing as a nondeterministic operator or a nondeterministic function. If some expression equals 42 today, then it will equal 42 tomorrow, and it will still equal 42 a million years from tomorrow. The specification

$$(x = \text{choose } n : n \in Nat) \land \Box[x' = \text{choose } n : n \in Nat]_x$$

allows only a single behavior—one in which $x$ always equals $\text{choose } n : n \in Nat$, which is some particular, unspecified natural number. It is very different from the specification

$$(x \in Nat) \land \Box[x' \in Nat]_x$$

The choose operator is known to logicians as Hilbert’s $\varepsilon$ [4].
that allows all behaviors in which $x$ is always a natural number—possibly a different number in each state. This specification is highly nondeterministic, allowing lots of different behaviors.
Chapter 7

Writing a Specification—Some Advice

You have now learned all you need to know about TLA+ to write your own specifications. Here are a few additional hints to help you get started.

7.1 Why to Specify

Specifications are written to help eliminate errors. Writing a specification requires effort; the benefits it provides must be worth that effort. There are several benefits:

• Writing a TLA+ specification can help the design process. Having to describe a design precisely often reveals problems—subtle interactions and “corner cases” that are easily overlooked. These problems are easier to correct when discovered in the design phase rather than after implementation has begun.

• A TLA+ specification can provide a clear, concise way of communicating a design. It helps ensure that the designers agree on what they have designed, and it provides a valuable guide to the engineers who implement and test the system. It may also help users understand the system.

• A TLA+ specification is a formal description to which tools can be applied to help find errors in the design and to help in testing the system. Some tools for TLA+ specifications are being built.

Whether these benefits justify the effort of writing the specification depends on the nature of the project. Specification is not an end in itself; it is just one of many tools that an engineer should be able to use when appropriate.
CHAPTER 7. WRITING A SPECIFICATION–SOME ADVICE

7.2 What to Specify

Although we talk about specifying a system, that’s not what we do. A specification is a mathematical model of a particular view of some part of a system. When writing a specification, the first thing you must choose is exactly what part of the system you want to describe. Sometimes the choice is obvious; often it isn’t. The cache-coherence protocol of a real multiprocessor computer may be intimately connected with how the processors execute instructions. Finding an abstraction that describes the coherence protocol while suppressing the details of instruction execution may be difficult. It may require defining an interface between the processor and the memory that doesn’t exist in the actual system design.

Remember that the purpose of a specification is to help avoid errors. You should specify those parts of the system for which a specification is most likely to reveal errors. TLA+ is particularly effective at revealing concurrency errors—ones that arise through the interaction of asynchronous components. So, you should concentrate your efforts on the parts of the system that are most likely to have such errors.

7.3 The Grain of Atomicity

After choosing what part of the system to specify, you must choose the specification’s level of abstraction. The most important aspect of the level of abstraction is the grain of atomicity, the choice of what system changes are represented as a single step of a behavior. Sending a message in an actual system involves multiple suboperations, but we usually represent it as a single step. On the other hand, the sending of a message and its receipt are usually represented as separate steps when specifying a distributed system.

The same sequence of system operations is represented by a shorter sequence of steps in a coarser-grained representation than in a finer-grained one. This almost always makes the coarser-grained specification simpler than the finer-grained one. However, the finer-grained specification more accurately describes the behavior of the actual system. A coarser-grained specification may fail to reveal important details of the system.

There is no simple rule for deciding on the grain of atomicity. However, there is one way of thinking about the granularity that can help. To describe it, we need the TLA+ action-composition operator “.”. If $A$ and $B$ are actions, then the action $A·B$ is executed by first executing $A$ then $B$ in a single step. More precisely, $A·B$ is the action defined by letting $s → t$ be an $A·B$ step if there exists a state $u$ such that $s → u$ is an $A$ step and $u → t$ is a $B$ step.

When determining the grain of atomicity, we must decide whether to represent the execution of an operation as a single step or as a sequence of steps, each
corresponding to the execution of a suboperation. Let’s consider the simple case of an operation consisting of two suboperations that are executed sequentially, where those suboperations are described by the two actions $R$ and $L$. (Executing $R$ enables $L$ and disables $R$.) When the operation’s execution is represented by two steps, each of those steps is an $R$ step or an $L$ step. The operation is then described with the action $R \lor L$. When its execution is represented by a single step, the operation is described with the action $R \cdot L$.¹ Let $S_2$ be the finer-grained specification in which the operation is executed in two steps, and let $S_1$ be the coarser-grained specification in which it is executed as a single $R \cdot L$ step. To choose the grain of atomicity, we must choose whether to take $S_1$ or $S_2$ as the specification. Let’s examine the relation between the two specifications.

We can transform any behavior $\sigma$ satisfying $S_1$ into a behavior $\bar{\sigma}$ satisfying $S_2$ by replacing each step $s \xrightarrow{R \cdot L} t$ with the pair of steps $s \xrightarrow{R} u \xrightarrow{L} t$, for some state $u$. If we regard $\sigma$ as being equivalent to $\bar{\sigma}$, then we can regard $S_1$ as being a strengthened version of $S_2$—one that allows fewer behaviors. Specification $S_1$ requires that each $R$ step be followed immediately by an $L$ step, while $S_2$ allows behaviors in which other steps come between the $R$ and $L$ steps. To choose the appropriate grain of atomicity, we must decide whether those additional behaviors allowed by $S_2$ are important.

The additional behaviors allowed by $S_2$ are not important if the actual system executions they describe are also described by behaviors allowed by $S_1$. So, we can ask whether each behavior $\tau$ satisfying $S_2$ has a corresponding behavior $\bar{\tau}$ satisfying $S_1$ that is, in some sense, equivalent to $\tau$. One way to construct $\bar{\tau}$ from $\tau$ is to transform a sequence of steps

\begin{equation}
(7.1) \quad s \xrightarrow{R} u_1 \xrightarrow{A_1} u_2 \xrightarrow{A_2} u_3 \ldots \xrightarrow{A_n} u_{n+1} \xrightarrow{L} t
\end{equation}

into the sequence

\begin{equation}
(7.2) \quad s \xrightarrow{A_1} v_1 \ldots v_{k-2} \xrightarrow{A_k} v_{k-1} \xrightarrow{R} v_k \xrightarrow{L} v_{k+1} \xrightarrow{A_{k+1}} v_{k+2} \ldots v_{n+1} \xrightarrow{A_n} t
\end{equation}

where the $A_i$ are other system actions that can be executed between the $R$ and $L$ steps. Both sequences start in state $s$ and end in state $t$, but the intermediate states may be different.

When is such a transformation possible? An answer can be given in terms of commutativity relations. We say that actions $A$ and $B$ commute if performing them in either order produces the same result. Formally, $A$ and $B$ commute iff $A \cdot B$ is equivalent to $B \cdot A$. A simple sufficient condition for commutativity is that two actions commute if they change the values of different variables and neither enables or disables the other. It’s not hard to see that we can transform (7.1) to (7.2) in the following two cases:

¹We actually describe the operation with an ordinary action, like the ones we’ve been writing, that is equivalent to $R \cdot L$. The operator $\"\cdot\"$ rarely appears in an actual specification. If you’re ever tempted to use it, look for a better way to write the specification; you can probably find one.
• $R$ commutes with each $A_i$. (In this case, $k = n$.)

• $L$ commutes with each $A_i$. (In this case, $k = 0$.)

In general, if an operation consists of a sequence of $m$ subactions, we must decide whether to choose the finer-grained representation $O_1 \lor O_2 \lor \ldots \lor O_m$ or the coarser-grained one $O_1 \cdot O_2 \cdots O_m$. The generalization of the transformation from (7.1) to (7.2) is one that transforms an arbitrary behavior satisfying the finer-grained specification into one in which the sequence of $O_1$, $O_2$, $\ldots$, $O_m$ steps come one right after the other. Such a transformation is possible if all but one of the actions $O_i$ commute with every other system action. Commutativity can be replaced by weaker conditions, but it is the most common case.

By commuting actions and replacing a sequence $s \xrightarrow{O_1} \cdots \xrightarrow{O_m} t$ of steps by a single $O_1 \cdots O_m$ step, you may be able to transform any behavior of a finer-grained specification into a corresponding behavior of a coarser-grained one. But that doesn’t mean that the coarser-grained specification is just as good as the finer-grained one. The sequences (7.1) and (7.2) are not the same, and a sequence of $O_i$ steps is not the same as a single $O_1 \cdots O_m$ step. Whether you can consider the transformed behavior to be equivalent to the original one, and use the coarser-grained specification, depends on the particular system you are specifying and on the purpose of the specification. Understanding the relation between finer- and coarser-grained specifications can help you choose between them; it won’t make the choice for you.

### 7.4 The Data Structures

Another aspect of a specification’s level of abstraction is the accuracy with which it describes the system’s data structures. For example, should the specification of a programming interface describe the actual layout of a procedure’s arguments in memory, or should the arguments be represented more abstractly?

To answer such a question, you must remember that the purpose of the specification is to help catch errors. A precise description of the layout of procedure arguments will help prevent errors caused by misunderstandings about that layout, but at the cost of complicating the programming interface’s specification. The cost is justified only if such errors are likely to be a real problem and the TLA+ specification provides the best way to avoid them.

If the purpose of the specification is to catch errors caused by the asynchronous interaction of concurrently executing components, then detailed descriptions of data structures will be a needless complication. So, you will probably want to use high-level, abstract descriptions of the system’s data structures in the specification. For example, to specify a program interface, you might introduce constant parameters to represent the actions of calling and return-
ing from a procedure—parameters analogous to Send and Reply of the memory interface described in Section 5.1 (page 45).

### 7.5 Writing the Specification

Once you've chosen the part of the system to specify and the level of abstraction, you're ready to start writing the TLA+ specification. We've already seen how this is done; let's review the steps.

First, pick the variables and define the type invariant and initial predicate. In the course of doing this, you will determine the constant parameters and assumptions about them that you need. You may also have to define some additional constants.

Next, write the next-state action, which forms the bulk of the specification. Sketching a few sample behaviors may help you get started. You must first decide how to decompose the next-state action as the disjunction of actions describing the different kinds of system operations. You then define those actions. The goal is to make the action definitions as compact and easy to read as possible. This requires carefully structuring them. One way to reduce the size of a specification is to define state predicates and state functions that are used in several different action definitions. When writing the action definitions, you will determine which of the standard modules you will need to use and add the appropriate `extends` statement. You may also have to define some constant operators for the data structures that you are using.

You must now write the temporal part of the specification. (If you want to specify liveness properties, you have to choose the fairness conditions, as described below in Chapter . You then combine the initial predicate, next-state action, and any fairness conditions you've chosen into the definition of a single temporal formula that is the specification.

Finally, you can assert theorems about the specification. If nothing else, you may want to add a type-correctness theorem.

### 7.6 Some Further Hints

Here are a few miscellaneous suggestions that may help you write better specifications.

**Don't be too clever.**

Cleverness can make a specification hard to read—and even wrong. The formula \( q = (h') \circ q' \) may look like a nice, short way of writing:

\[
(7.3) \quad (h' = Head(q)) \land (q' = Tail(q))
\]
But not only is \( q = h \circ q' \) harder to understand than (7.3), it’s also wrong. We don’t know what \( a \circ b \) equals if \( a \) and \( b \) are not both sequences, so we don’t know whether \( h_0 = \text{Head}(q) \) and \( q_0 = \text{Tail}(q) \) are the only values of \( h' \) and \( q' \) that satisfy \( q = (h') \circ q' \). There could be other values of \( h' \) and \( q' \), which are not sequences, that satisfy the formula.

**A type invariant is not an assumption.**

Type invariance is a property of a specification, not an assumption. When writing a specification, we usually define a type invariant. But, that’s just a definition; a definition is not an assumption. Suppose you define a type invariant that asserts that a variable \( n \) is of type \( \text{Nat} \). You may be tempted to then think that a conjunct \( n' > 7 \) in an action asserts that \( n' \) is a natural number greater than 7. It doesn’t. The formula \( n' > 7 \) asserts only that \( n' > 7 \). It is satisfied if \( n' = \sqrt{96} \) as well as if \( n' = 8 \). Since we don’t know whether or not “abc” \( > 7 \) is true, it might be satisfied if \( n' = \text{“abc”} \). The meaning of the formula is not changed just because you’ve defined a type invariant that asserts \( n \in \text{Nat} \).

In general, you may want to describe the new value of a variable \( x \) by asserting some property of \( x' \). However, the next-state relation should imply that \( x' \) is an element of some suitable set. For example, a specification might define:

\[
\begin{align*}
\text{Action1} & \triangleq (n' > 7) \land \ldots \\
\text{Action2} & \triangleq (n' \leq 6) \land \ldots \\
\text{Next} & \triangleq (n' \in \text{Nat}) \land (\text{Action1} \lor \text{Action2})
\end{align*}
\]

**Don’t assume values that look different are unequal.**

The rules of TLA do not imply that \( 1 \neq \text{“a”} \). If the system can send a message that is either a string or a number, represent the message as a record with a type and value field—for example,

\[
[\text{type} \mapsto \text{“String”}, \text{value} \mapsto \text{“a”}] \quad \text{or} \quad [\text{type} \mapsto \text{“Nat”}, \text{value} \mapsto 1]
\]

**Move quantification to the outside.**

Specifications are usually easier to read if \( \exists \) is moved outside disjunctions and \( \forall \) is moved outside conjunctions. For example, instead of:

\[
\begin{align*}
\text{Up} & \triangleq \exists e \in \text{Elevator} : \ldots \\
\text{Down} & \triangleq \exists e \in \text{Elevator} : \ldots \\
\text{Move} & \triangleq \text{Up} \lor \text{Down}
\end{align*}
\]

\(^2\) An alternative approach is to define \( \text{Next} \) to equal \( \text{Action1} \lor \text{Action2} \) and to let the specification be \( \text{Init} \land \Box[\text{Next}] \ldots \land \Box(\exists n \in \text{Nat}) \). But, it’s usually better to stick to the simple form \( \text{Init} \land \Box[\text{Next}] \ldots \) for specifications.
it’s usually better to write:

\[
\begin{align*}
Up(e) & \triangleq \ldots \\
Down(e) & \triangleq \ldots \\
Move & \triangleq \exists e \in \text{Elevator} : Up(e) \lor Down(e)
\end{align*}
\]

Write comments as comments.

Don’t put comments into the specification itself. I have seen people write things like the following action definition:

\[
A \triangleq \lor \land x \geq 0 \\
\land \ldots \\
\lor \land x < 0 \\
\land \text{FALSE}
\]

The second disjunct is meant to indicate that the writer intended \(A\) not to be enabled when \(x < 0\). But that disjunct is completely redundant, since \(F \land \text{FALSE}\) equals \(\text{FALSE}\), and \(F \lor \text{FALSE}\) equals \(F\), for any formula \(F\). So the second disjunct of the definition serves only as a form of comment. It’s better to write:

\[
A \triangleq \land x \geq 0 \\
\land \ldots \\
\text{A is not enabled if } x < 0
\]

7.7 When and How to Specify

Specifications are often written later than they should be. Engineers are usually under severe time constraints, and they may feel that writing a specification will slow them down. Only after a design has become so complex that they need help understanding it do engineers think about writing a precise specification.

Writing a specification helps you think clearly. Thinking clearly is hard; we can use all the help we can get. Making specification part of the design process can improve the design.

I have described how to write a specification assuming that the system design already exists. But it’s better to write the specification as the system is being designed. The specification will start out being incomplete and probably incorrect. For example, an initial specification of the write-through cache of Section 5.6 (page 54) might include the definition:

\[
\begin{align*}
\text{RdMiss}(p) & \triangleq \text{Enqueue a request to write value from memory to } p\text{'s cache.} \\
& \text{Some enabling condition must be conjoined here.} \\
\land \text{memQ'} = \text{Append}(\text{memQ}, \text{buf}[p]) & \text{Append request to memQ.} \\
\land \text{ctl}' = [\text{ctl} \text{ EXCEPT ![p] = "?"}] & \text{Set ctl}[p] \text{ to value to be determined later.} \\
\land \text{UNCHANGED } \langle \text{memInt}, \text{mem}, \text{buf}, \text{cache} \rangle
\end{align*}
\]
Some system functionality will at first be omitted; it can be included later by adding new disjuncts to the next-state action. Tools can be applied to these preliminary specifications to help find design errors.
Part II

More Advanced Topics
Chapter 8

Liveness and Fairness

The specifications we have written so far say what a system must not do. The clock must not advance from 11 to 9; the receiver must not receive a message if the FIFO is empty. They don’t require that the system ever do anything. The clock need never tick; the sender need never send any messages. Our specifications have described what are called safety properties. If a safety property is violated, it is violated at some particular point in the behavior—by a step that advances the clock from 11 to 9, or that reads the wrong value from memory.

We now learn how to specify that something does happen: the clock keeps ticking; a value is eventually read from memory. We specify liveness properties, which cannot be violated at any particular instant. Only by examining an entire infinite behavior can we tell that the clock has stopped ticking, or that a message is never sent.

8.1 Temporal Formulas

To specify liveness properties we must learn to express them as temporal formulas. We now take a more rigorous look at what a temporal formula means.

Recall that a state assigns a value to every variable, and a behavior is an infinite sequence of states. A temporal formula is true or false of a behavior. Let $\sigma \models F$ be the truth value of the formula $F$ for the behavior $\sigma$, so $\sigma$ satisfies $F$ iff $\sigma \models F$ equals true. To define the meaning of a temporal formula $F$, we have to explain how to determine the value of $\sigma \models F$ for any behavior $\sigma$. For now, we consider only temporal formulas that don’t contain the temporal existential quantifier $\exists$.

It’s easy to define the meaning of a Boolean combination of temporal formulas in terms of the meanings of those formulas. The formula $F \land G$ is true of a behavior $\sigma$ iff both $F$ and $G$ are true of $\sigma$, and $\neg F$ is true of $\sigma$ iff $F$ is false for
\[\sigma: \quad \sigma \models (F \land G) \triangleq (\sigma \models F) \land (\sigma \models G) \quad \sigma \models \neg F \triangleq \neg (\sigma \models F)\]

This defines the meanings of \(\land\) and \(\neg\) as operators on temporal formulas. The meanings of the other Boolean operators are similarly defined. In the same way, we can define the ordinary predicate-logic quantifiers \(\forall\) and \(\exists\) as operators on temporal formulas—for example:

\[\sigma \models (\exists r : F) \triangleq \exists r : (\sigma \models F)\]

We will discuss quantifiers in Section 8.6. For now, we ignore quantification in temporal formulas.

All the temporal formulas not containing \(\exists\) that we’ve seen have been Boolean combinations of the following three simple kinds of formulas. (Recall the definitions of state function and state predicate on page 25 in Section 3.1.)

- A state predicate. It is interpreted as a temporal formula that is true of a behavior iff it is true in the first state of the behavior.

- A formula \(\Box P\), where \(P\) is a state predicate. It is true of a behavior iff \(P\) is true in every state of the behavior.

- A formula \(\Box [N]_v\), where \(N\) is an action and \(v\) is a state function. It is true of a behavior iff every successive pair of steps in the behavior is a \([N]_v\) step.

Since a state predicate is an action that contains no primed variables, we can both combine and generalize these three kinds of temporal formulas into the two kinds of formulas \(\Box\) and \(\hspace{1em} \Box\), where \(A\) is an action.

Generalizing from state functions, we interpret an arbitrary action \(A\) as a temporal formula by defining \(\sigma \models A\) to be true iff the first two states of \(\sigma\) are an \(A\) step. For any behavior \(\sigma\), let \(\sigma_0, \sigma_1, \ldots\) be the sequence of states that make up \(\sigma\). Then the meaning of an action \(A\) as a temporal formula is defined by letting \(\sigma \models A\) be true iff the pair \(\langle \sigma_0, \sigma_1 \rangle\) of states is an \(A\) step. We define \(\sigma \models \Box A\) to be true iff, for all \(n \in \text{Nat}\), the pair \(\langle \sigma_n, \sigma_{n+1} \rangle\) of states is an \(A\) step. We now generalize this to the definition of \(\sigma \models \Box F\) for an arbitrary temporal formula \(F\).

For any behavior \(\sigma\) and natural number \(n\), let \(\sigma^{+n}\) be the suffix of \(\sigma\) obtained by deleting its first \(n\) states:

\[\sigma^{+n} \triangleq \sigma_n, \sigma_{n+1}, \sigma_{n+2}, \ldots\]

The successive pair of states \(\langle \sigma_n, \sigma_{n+1} \rangle\) of \(\sigma\) is the first pair of states of \(\sigma^{+n}\), and \(\langle \sigma_n, \sigma_{n+1} \rangle\) is an \(A\) step iff \(\sigma^{+n}\) satisfies \(A\). In other words:

\[\langle \sigma \models \Box A \rangle \equiv \forall n \in \text{Nat} : \sigma^{+n} \models A\]
So, we can generalize the definition of $\sigma \models \Box A$ to
\[ \sigma \models \Box F \equiv \forall n \in \text{Nat} : \sigma^{n} \models F \]
for any temporal formula $F$. In other words, $\sigma$ satisfies $\Box F$ iff every suffix $\sigma^{n}$ of $\sigma$ satisfies $F$. This defines the temporal operator $\Box$.

We have now defined the meaning of any temporal formula built from actions (including state predicates), Boolean operators, and the $\Box$ operator. For example:
\[ \sigma \models \Box((x = 1) \Rightarrow \Box(y > 0)) \]
\[ \equiv \forall n \in \text{Nat} : \sigma^{n} \models ((x = 1) \Rightarrow \Box(y > 0)) \]  
By the meaning of $\Box$.
\[ \equiv \forall n \in \text{Nat} : (\sigma^{n} \models (x = 1)) \Rightarrow (\sigma^{n} \models \Box(y > 0)) \]  
By the meaning of $\Rightarrow$.
\[ \equiv \forall n \in \text{Nat} : (\sigma^{n} \models (x = 1)) \Rightarrow (\forall m \in \text{Nat} : (\sigma^{n+m} \models (y > 0))) \]  
By the meaning of $\Box$.

Thus, $\sigma \models \Box((x = 1) \Rightarrow \Box(y > 0))$ is true iff, for all $n \in \text{Nat}$, if $x = 1$ is true in state $\sigma_{n}$, then $y > 0$ is true in all states $\sigma_{n+m}$ with $m \geq 0$.

We saw in Section 2.2 that a specification should allow stuttering steps—ones that leave unchanged all the variables appearing in the formula. A stuttering step represents a change only to some part of the system not described by the formula; adding it to the behavior should not affect the truth of the formula. We say that a formula $F$ is invariant under stuttering\(^1\) if adding a stuttering step to a behavior $\sigma$ does not affect whether $F$ is true of $\sigma$. (This implies that removing a stuttering step from $\sigma$ also does not affect the truth of $\sigma \models F$.) A sensible formula should be invariant under stuttering. There’s no point writing formulas that aren’t sensible, so TLA$^{+}$ allows you to write only temporal formulas that are invariant under stuttering.

An arbitrary action, viewed as a temporal formula, is not invariant under stuttering. The action $x' = x + 1$ is true for a behavior in which $x$ is incremented by 1 in the first step; adding an initial stuttering step makes it false. A state predicate is invariant under stuttering, since its truth depends only on the first state of a behavior, and adding a stuttering step doesn’t change the first state. The formula $\Box[N]v$ is also invariant under stuttering, for any action $N$ and state function $v$. It’s not hard to see that invariance under stuttering is preserved by $\Box$ and by the Boolean operators. So, state predicates, formulas of the form $\Box[N]v$, and all formulas obtainable from them by applying $\Box$ and Boolean operators are invariant under stuttering. For now, let’s take these to be our temporal formulas. (Later, we’ll add quantification.)

To understand temporal formulas intuitively, think of $\sigma_{n}$ as the state of the universe at time instant $n$ during the behavior $\sigma$.\(^2\) For any state predicate

\(^1\)This is a completely new sense of the word invariant; it has nothing to do with the concept of invariance discussed already.

\(^2\)It is because we think of $\sigma_{n}$ as the state at time $n$, and because we usually measure time starting from 0, that I start numbering the states of a behavior with 0 rather than 1.
By definition of $P$, the expression $\sigma^+ n \models P$ asserts that $P$ is true at time instant $n$. Thus, $\Box((x = 1) \Rightarrow \Box(y > 0))$ asserts that any time $x = 1$ is true, $y > 0$ is true from then on. For an arbitrary temporal formula $F$, we also interpret $\sigma^+ n \models F$ as the assertion that $F$ is true at time instant $n$. The formula $\Box F$ then asserts that $F$ is true at all times. We can therefore read $\Box$ as always or henceforth or from then on.

We now examine five especially important classes of formulas that are constructed from arbitrary temporal formulas $F$ and $G$. We introduce new operators for expressing the first three.

$\Diamond F$ is defined to equal $\neg \Box \neg F$. It asserts that $F$ is not always false, which means that $F$ is true at some time:

$\sigma \models \Diamond F$

$\equiv \sigma \models \neg \Box \neg F$  

By definition of $\Diamond$.

$\equiv \neg (\sigma \models \neg \Box \neg F)$  

By the meaning of $\neg$.

$\equiv \neg (\forall n \in \text{Nat} : \sigma^+ n \models \neg F)$  

By the meaning of $\Box$.

$\equiv \neg (\forall n \in \text{Nat} : \neg (\sigma^+ n \models F))$  

By the meaning of $\neg$.

$\equiv \exists n \in \text{Nat} : \sigma^+ n \models F$  

Because $\forall \neg$ is equivalent to $\exists$.

We usually read $\Diamond$ as eventually, taking eventually to include now.

$F \leadsto G$ is defined to equal $\Box(F \Rightarrow \Diamond G)$. The same kind of calculation we’ve done above shows

$\sigma \models (F \leadsto G)$

$\equiv \forall n \in \text{Nat} : (\sigma^+ n \models F) \Rightarrow (\exists m \in \text{Nat} : (\sigma^+(n+m) \models G))$

The formula $F \leadsto G$ asserts that whenever $F$ is true, $G$ is eventually true—that is, true now or at some later time. We read $\leadsto$ as leads to.

$\Diamond \langle A \rangle_v$ is defined to equal $\neg \Box[\neg A]_v$, where $A$ is an action and $v$ a state function. It asserts that not every step is a $(\neg A) \lor (v' = v)$ step, so some step is a $\neg((\neg A) \lor (v' = v))$ step. But $\neg((\neg A) \lor (v' = v))$ is equivalent to $A \land (v' \neq v)$, so $\Diamond \langle A \rangle_v$ asserts that some step is an $A \land (v' \neq v)$ step. We define $\langle A \rangle_v$ to equal $A \land (v' \neq v)$, so $\Diamond \langle A \rangle_v$ asserts that eventually an $\langle A \rangle_v$ step occurs. We think of $\Diamond \langle A \rangle_v$ as the formula obtained by applying the operator $\Diamond$ to $\langle A \rangle_v$, although technically it’s not because $\langle A \rangle_v$ isn’t a temporal formula.

$\Box \Diamond F$ asserts that at all times, $F$ is true then or at some later time. For time instant 0, this implies that $F$ is true at some time instant $n_0 \geq 0$. For time instant $n_0 + 1$, it implies that $F$ is true at some time instant $n_1 \geq n_0 + 1$. For time instant $n_1 + 1$, it implies that $F$ is true at some time instant $n_2 \geq n_1 + 1$. Continuing the process, we see that $F$ is true at an infinite sequence of time instants $n_0, n_1, n_2, \ldots$. So, $\Box \Diamond F$ asserts that $F$
is infinitely often true. In particular, □◊⟨A⟩ₐ asserts that infinitely many ⟨A⟩ₐ steps occur.

◇□F asserts that eventually (at some time), F becomes true and remains true thereafter. In other words, ◇□F asserts that F is eventually always true.

In particular, ◇□[N]ₐ asserts that eventually, every step is a [N]ₐ step.

The operators □ and ◇ have higher precedence (bind more tightly) than the Boolean operators, so ◇F ∨ □G means (◇F) ∨ (□G). The operator ⇒ has the same precedence as ⊢.

## 8.2 Weak Fairness

Using the temporal operators □ and ◇, it’s easy to specify liveness properties. For example, consider the hour-clock specification of module `HourClock` in Figure 2.1 on page 20. We can require that the clock never stops by asserting that there must be infinitely many HCₐₐ steps. This is expressed by the formula □◊⟨HCₐₐ⟩ₐₐ. (The (⟨⟩ₐₐ is needed to satisfy the syntax rules for temporal formulas; it’s discussed in the next paragraph.) So, formula HC ∧ □◊⟨HCₐₐ⟩ₐₐ specifies a clock that never stops.

The syntax rules of TLA require us to write □◊⟨HCₐₐ⟩ₐₐ instead of the more obvious formula □◊HCₐₐ. These rules are needed to guarantee that every syntactically correct TLA formula is invariant under stuttering. In a behavior satisfying HC, an HCₐₐ step necessarily changes hr, so it is necessarily an ⟨HCₐₐ⟩ₐₐ step. Hence, HC ∧ □◊⟨HCₐₐ⟩ₐₐ is equivalent to the (illegal) formula HC ∧ □◊HCₐₐ.

In a similar fashion, most of the actions we define do not allow stuttering steps. When we write ⟨A⟩ₐ, for some action A, the ⟨⟩ₐ is usually needed only to satisfy the syntax rules. To avoid having to think about which variables A actually changes, we generally take the subscript v to be the tuple of all variables, which is changed if any variable changes. I will usually ignore the angle brackets and subscripts in informal discussions, and will describe □◊⟨HCₐₐ⟩ₐₐ as the assertion that there are infinitely many HCₐₐ steps.

Let’s now modify specification `Spec` of module `Channel` (Figure 3.2 on page 30) to require that every value sent is eventually received. We do this by conjoining a liveness condition to `Spec`. The analog of the liveness condition for the clock would be □◊⟨Rcv⟩ₐ₇⅛, which asserts that there are infinitely many Rcv steps. However, this would also require that infinitely many values are sent, and we don’t want to make that requirement. In fact, we want to permit behaviors in which no value is ever sent, so no value is ever received. We just want to require that, if a value is ever sent, then it is eventually received.

It’s enough to require only that the next value to be received always is eventually received, since this implies that all values sent are eventually received.
More precisely, we require that it’s always the case that, if there is a value to be received, then the next value to be received eventually is received. The next value is received by a $Rcv$ step, so the requirement is:

\[ \square (\text{There is an unreceived value} \implies \Diamond \langle Rcv \rangle_{chan}) \]

There is an unreceived value iff action $Rcv$ is enabled. (Recall that $Rcv$ is enabled in a state if it is possible to take a $Rcv$ step from that state.) TLA\textsuperscript{+} defines $\text{Enabled } A$ to be the predicate that is true iff action $A$ is enabled. The liveness condition can then be written as:

\[ (\square (\text{Enabled } \langle Rcv \rangle_{chan} \implies \Diamond \langle Rcv \rangle_{chan}) \]

In the $\text{Enabled}$ formula, it doesn’t matter if we write $Rcv$ or $\langle Rcv \rangle_{chan}$. We add the angle brackets so the two actions appearing in the formula are the same.

Because $\langle HCnxt \rangle_{hr}$ is always enabled during any behavior satisfying $HC$, we can rewrite the liveness condition $\square \Diamond \langle HCnxt \rangle_{hr}$ for the hour clock as:

\[ \square (\text{Enabled } \langle HCnxt \rangle_{hr} \implies \Diamond \langle HCnxt \rangle_{hr}) \]

This suggests a general liveness condition on an action $A$:

\[ \square (\text{Enabled } \langle A \rangle_{v} \implies \Diamond \langle A \rangle_{v}) \]

This formula asserts that if $A$ ever becomes enabled, then an $A$ step will eventually occur—even if $A$ remains enabled for only a fraction of a nanosecond, and is never again enabled. The obvious difficulty of physically implementing such a requirement suggests that it’s too strong. Instead, we define the weaker formula $WF_{v}(A)$ to equal:

\[ (\square \square \text{Enabled } \langle A \rangle_{v} \implies \Diamond \langle A \rangle_{v}) \]

This formula asserts that if $A$ ever becomes forever enabled, then an $A$ step must eventually occur. $WF$ stands for Weak Fairness, and the condition $WF_{v}(A)$ is called weak fairness on $A$. Here are two formulas that are each equivalent to (8.2):

\begin{align*}
8.3 & \quad \square \Diamond (\neg \text{Enabled } \langle A \rangle_{v}) \lor \square \Diamond \langle A \rangle_{v} \\
8.4 & \quad \Diamond \square (\text{Enabled } \langle A \rangle_{v}) \implies \square \Diamond \langle A \rangle_{v}
\end{align*}

These three formulas can be expressed in English as:

8.2. It’s always the case that, if $A$ is enabled forever, then an $A$ step eventually occurs.

8.3. $A$ is infinitely often disabled, or infinitely many $A$ steps occur.

\[ ^{3} \square (F \implies \Diamond G) \text{ equals } F \leadsto G, \text{ so we could write this formula more compactly with } \leadsto . \]

However, it is more convenient to keep it in the form $\square (F \implies \Diamond G)$.
8.2. WEAK FAIRNESS

8.4. If $A$ is eventually enabled forever, then infinitely many $A$ steps occur.

The equivalence of these three formulas isn’t obvious. Here’s a proof that (8.2) is equivalent to (8.3), using some simple tautologies. Studying this proof, and these tautologies, will help you understand how to write liveness conditions.

\[
\Box(\Box \text{Enabled} \langle A \rangle_v) \Rightarrow \Diamond \langle A \rangle_v
\]

\[\equiv \Box (\lnot \Box \text{Enabled} \langle A \rangle_v \lor \Diamond \langle A \rangle_v)\]  
Because $(F \Rightarrow G) \equiv (\lnot F \lor G)$.

\[\equiv \Box (\Diamond \lnot \Box \text{Enabled} \langle A \rangle_v \lor \Diamond \langle A \rangle_v)\]  
Because $\Box \lnot F \equiv \lnot \Box F$.

\[\equiv \Box (\lnot \Box \text{Enabled} \langle A \rangle_v \lor \Diamond \langle A \rangle_v)\]  
Because $\Diamond \lnot (F \lor G) \equiv \Diamond F \lor \Diamond G$.

\[\equiv \Box (\Diamond \lnot \Box \text{Enabled} \langle A \rangle_v \lor \Diamond \langle A \rangle_v)\]  
Because $\Diamond \lnot (F \lor G) \equiv \Box \lnot F \lor \Box \lnot G$.

The equivalence of (8.3) and (8.4) is proved as follows

\[
\Box \Diamond (\lnot \Box \text{Enabled} \langle A \rangle_v) \Rightarrow \Box \Diamond \langle A \rangle_v
\]

\[\equiv \lnot \Box \Diamond \Box \text{Enabled} \langle A \rangle_v \lor \Box \Diamond \langle A \rangle_v\]  
Because $\Box \lnot F \Rightarrow \Box \lnot F$  
Because $(F \Rightarrow G) \equiv (\lnot F \lor G)$.

\[\equiv \Diamond \Box \text{Enabled} \langle A \rangle_v \Rightarrow \Box \Diamond \langle A \rangle_v\]  
Because $\Diamond F \Rightarrow \Box \lnot F$  
Because $\Box \lnot F \Rightarrow \lnot \Box F$.

\[
\Box \Diamond (\lnot \Box \text{Enabled} \langle A \rangle_v) \Rightarrow \Box \Diamond \langle A \rangle_v
\]

We now show that the liveness conditions for the hour clock and the channel can be written as weak fairness conditions.

First, consider the hour clock. In any behavior satisfying its safety specification $HC$, an $\langle HC_{nxt} \rangle_{hr}$ step is always enabled, so $\Box \Diamond \langle HC_{nxt} \rangle_{hr}$ equals true. Hence, $HC$ implies that WF$_{hr}(HC_{nxt})$ is equivalent to $\Box \Diamond \langle HC_{nxt} \rangle_{hr}$, our liveness condition for the hour clock.

Now, consider the liveness condition (8.1) for the channel. By (8.3), weak fairness on $Rcv$ asserts that either (a) $Rcv$ is disabled infinitely often, or (b) infinitely many $Rcv$ steps occur (or both). Suppose $Rcv$ becomes enabled at some instant. In case (a), $Rcv$ must subsequently be disabled, which can occur only by a $Rcv$ step. Case (b) also implies that there is a subsequent $Rcv$ step. Weak fairness therefore implies that it’s always the case that if $Rcv$ is enabled, then a $Rcv$ step eventually occurs. A closer look at this reasoning reveals that it is an informal proof of:

\[
Spec \Rightarrow (WF_{chan}(Rcv) \Rightarrow \Box(\Diamond \langle Rcv \rangle_{chan} \Rightarrow \Diamond \langle Rcv \rangle_{chan}))
\]

Because $\Box F$ implies $F$, for any formula $F$, it’s not hard to check the truth of:

\[
\Box(\Diamond \langle Rcv \rangle_{chan} \Rightarrow \Diamond \langle Rcv \rangle_{chan}) \Rightarrow WF_{chan}(Rcv)
\]

Therefore, $Spec \land WF_{chan}(Rcv)$ is equivalent to the conjunction of $Spec$ and (8.1), so weak fairness of $Rcv$ specifies the same liveness condition as (8.1) for the channel.
8.3 Liveness for the Memory Specification

Let's now strengthen the memory specification with the liveness requirement that every request must receive a response. (We don’t require that a request is ever issued.) The liveness requirement is conjoined to the internal memory specification, formula $\mathit{ISpec}$ of module $\mathit{InternalMemory}$ (Figures 5.2 and 5.3 on pages 52–53).

We will express the liveness requirement in terms of weak fairness. This requires understanding when actions are enabled. The action $\mathit{Rsp}(p)$ is enabled only if the action

\begin{equation}
\mathit{Reply}(p, \mathit{buf}[p], \mathit{memInt}, \mathit{memInt}')
\end{equation}

is enabled. Recall that the operator $\mathit{Reply}$ is a constant parameter, declared in the $\mathit{MemoryInterface}$ module (Figure 5.1 on page 48). Without knowing more about this operator, we can’t say when action (8.5) is enabled.

Let’s assume that $\mathit{Reply}$ actions are always enabled. That is, for any processor $p$ and reply $r$, and any old value $\mathit{miOld}$ of $\mathit{memInt}$, there is a new value $\mathit{miNew}$ of $\mathit{memInt}$ such that $\mathit{Repl}(p, r, \mathit{miOld}, \mathit{miNew})$ is true. For simplicity, we just assume that this is true for all $p$ and $r$, and add the following assumption to the $\mathit{MemoryInterface}$ module:

\begin{align*}
\text{assume } & \forall p, r, \mathit{miOld} : \exists \mathit{miNew} : \mathit{Repl}(p, r, \mathit{miOld}, \mathit{miNew})
\end{align*}

We should also make a similar assumption for $\mathit{Send}$, but we don’t need it here.

We will subscript our weak-fairness formulas with the tuple of all variables, so let’s add the following definition to the $\mathit{InternalMemory}$ module:

\begin{align*}
\mathit{vars} & \equiv (\mathit{memInt}, \mathit{mem}, \mathit{ctl}, \mathit{buf})
\end{align*}

When processor $p$ issues a request, it enables the $\mathit{Do}(p)$ action, which remains enabled until a $\mathit{Do}(p)$ step occurs. The weak-fairness condition $\mathit{WF}_{\mathit{vars}}(\mathit{Do}(p))$ implies that this $\mathit{Do}(p)$ step must eventually occur. A $\mathit{Do}(p)$ step enables the $\mathit{Rsp}(p)$ action, which remains enabled until an $\mathit{Rsp}(p)$ step occurs. The weak-fairness condition $\mathit{WF}_{\mathit{vars}}(\mathit{Rsp}(p))$ implies that this $\mathit{Rsp}(p)$ step, which produces the desired response, must eventually occur. Hence, the requirement

\begin{equation}
\mathit{WF}_{\mathit{vars}}(\mathit{Do}(p)) \land \mathit{WF}_{\mathit{vars}}(\mathit{Rsp}(p))
\end{equation}

assures that every request issued by processor $p$ must eventually receive a reply.

We can rewrite condition (8.6) in the slightly simpler form of weak fairness on the action $\mathit{Do}(p) \lor \mathit{Rsp}(p)$. The disjunction of two actions is enabled iff one or both of them are enabled. A $\mathit{Req}(p)$ step enables $\mathit{Do}(p)$, thereby enabling $\mathit{Do}(p) \lor \mathit{Rsp}(p)$. The only $\mathit{Do}(p) \lor \mathit{Rsp}(p)$ step then possible is a $\mathit{Do}(p)$ step, which enables $\mathit{Rsp}(p)$ and hence $\mathit{Do}(p) \lor \mathit{Rsp}(p)$. At this point, the only $\mathit{Do}(p) \lor \mathit{Rsp}(p)$ step possible is a $\mathit{Rsp}(p)$ step, which disables $\mathit{Rsp}(p)$ and leaves $\mathit{Do}(p)$
disabled, hence disabling $Do(p) \lor Rsp(p)$. This all shows that (8.6) is equivalent to $WF_{vars}(Do(p) \lor Rsp(p))$, weak fairness on the single action $Do(p) \lor Rsp(p)$.

Weak fairness of $Do(p) \lor Rsp(p)$ guarantees that every request by processor $p$ receives a response. We want every request from every processor to receive a response. So, the liveness condition for the memory specification asserts weak fairness of $Do(p) \lor Rsp(p)$ for every processor $p$:

\[
\text{Liveness} \triangleq \forall p \in \text{Proc} : WF_{vars}(Do(p) \lor Rsp(p))
\]

The example of actions $Do(p)$ and $Rsp(p)$ raises the general question: when is the conjunction of weak fairness on actions $A_1, \ldots, A_n$ equivalent to weak fairness of their disjunction $A_1 \lor \ldots \lor A_n$? The general answer is complicated, but here’s a sufficient condition:

**WF Conjunction Rule** If $A_1, \ldots, A_n$ are actions such that, for any distinct $i$ and $j$, whenever action $A_i$ is enabled, action $A_j$ cannot become enabled until an $A_i$ step occurs, then $WF_v(A_1) \land \ldots \land WF_v(A_n)$ is equivalent to $WF_v(A_1 \lor \ldots \lor A_n)$.

This rule is stated rather informally. It can be interpreted as an assertion about a particular behavior $\sigma$, in which case its conclusion is

\[
\sigma \models (WF_v(A_1) \land \ldots \land WF_v(A_n)) \equiv WF_v(A_1 \lor \ldots \lor A_n)
\]

Alternatively, it can be formalized as the assertion that if a formula $\mathcal{F}$ implies the hypothesis, then $\mathcal{F}$ implies the equivalence of $WF_v(A_1) \land \ldots \land WF_v(A_n)$ and $WF_v(A_1 \lor \ldots \lor A_n)$.

Conjunction and disjunction are special cases of universal and existential quantification, respectively. For example, $A_1 \lor \ldots \lor A_n$ is equivalent to $\exists i \in 1 \ldots n : A_i$.

So, we can trivially restate the WF Conjunction Rule as a condition on when $\forall i \in S : WF_v(A_i)$ and $WF_v(\exists i \in S : A_i)$ are equivalent, for a finite set $S$. The resulting rule is actually valid for any set $S$:

**WF Quantifier Rule** If the $A_i$ are actions, for all $i \in S$, such that, for any distinct $i$ and $j$ in $S$, whenever action $A_i$ is enabled, action $A_j$ cannot become enabled until an $A_i$ step occurs, then $\forall i \in S : WF_v(A_i)$ and $WF_v(\exists i \in S : A_i)$ are equivalent.

### 8.4 Strong Fairness

Formulations (8.3) and (8.4) of $WF_v(A)$ contain the operators infinitely often ($\Box \Diamond$) and eventually always ($\Diamond \Box$). Eventually always is stronger than (implies)

\[\text{Although we haven’t yet discussed quantification in temporal formulas, the meaning of the formula } \forall p \in \text{Proc} : \ldots \text{ should be clear.}\]
CHAPTER 8. LIVENESS AND FAIRNESS

infinitely often. We define $\text{SF}_v(A)$, strong fairness of action $A$, to be either of the following equivalent formulas:

\begin{align*}
\text{(8.7)} \quad & \Diamond \Diamond (\neg \text{Enabled } (A)_v) \lor \Box \Diamond (A)_v \\
\text{(8.8)} \quad & \Box \Diamond \text{Enabled } (A)_v \Rightarrow \Box \Diamond (A)_v
\end{align*}

Intuitively, these two formulas assert:

\begin{align*}
\text{(8.7)} \quad & \text{$A$ is eventually disabled forever, or infinitely many $A$ steps occur.} \\
\text{(8.8)} \quad & \text{If $A$ is infinitely often enabled, then infinitely many $A$ steps occur.}
\end{align*}

The proof that these two formulas are equivalent is similar to the proof of equivalence of (8.3) and (8.4).

The analogs of the WF Conjunction and WF Quantifier Rules (page 93) hold for strong fairness—for example:

**SF Conjunction Rule** If $A_1, \ldots, A_n$ are actions such that, for any distinct $i$ and $j$, whenever action $A_i$ is enabled, action $A_j$ cannot become enabled until an $A_i$ step occurs, then $\text{SF}_v(A_1) \land \ldots \land \text{SF}_v(A_n)$ is equivalent to $\text{SF}_v(A_1 \lor \ldots \lor A_n)$.

It’s not hard to see that strong fairness is stronger than weak fairness—that is, $\text{SF}_v(A)$ implies $\text{WF}_v(A)$, for any $v$ and $A$. We can express weak and strong fairness as follows.

- Weak fairness of $A$ asserts that an $A$ step must eventually occur if $A$ is continuously enabled.
- Strong fairness of $A$ asserts that an $A$ step must eventually occur if $A$ is continually enabled.

*Continuously* means without interruption. *Continually* means repeatedly, possibly with interruptions.

Strong fairness need not be strictly stronger than weak fairness. Weak and strong fairness of an action $A$ are equivalent iff $A$ infinitely often disabled implies that either $A$ is eventually always disabled, or infinitely many $A$ steps occur. This is expressed formally by the tautology:

\[(\text{WF}_v(A) \equiv \text{SF}_v(A)) \equiv (\Box \Diamond (\neg \text{Enabled } (A)_v) \Rightarrow \Box \Diamond (\neg \text{Enabled } (A)_v) \lor \Box \Diamond (A)_v)\]

In the channel example, weak and strong fairness of $\text{Rcv}$ are equivalent because $\text{Spec}$ implies that, once enabled, $\text{Rcv}$ can be disabled only by a $\text{Rcv}$ step; so if it is disabled infinitely often, then it either eventually remains disabled forever, or else it is disabled infinitely often by $\text{Rcv}$ steps.

Strong fairness can be more difficult to implement than weak fairness, and it is a less common requirement. A strong fairness condition should be used in a
specification only if it is needed. When strong and weak fairness are equivalent, the fairness property should be written as weak fairness.

Liveness properties can be subtle. Expressing them with ad hoc temporal formulas can lead to errors. We will specify liveness as the conjunction of fairness properties whenever possible—and it almost always is possible. Having a uniform way of expressing liveness makes specifications easier to understand. Section 8.7.2 discusses an even more compelling reason for using fairness to specify liveness.

8.5 Liveness for the Write-Through Cache

Let’s now add liveness to the write-through cache, specified in Figures 5.5–5.7 on pages 56–58. We want our specification to guarantee that every request eventually receives a response, without requiring that any requests are issued. This requires fairness on all the actions that make up the next-state action Next except the Req(p) action (which issues a request) and the Evict(p, a) action (which evicts an address from the cache). If any other action were ever enabled without being executed, then some request might not generate a response—except for one special case. If the memQ queue contains only write requests, and memQ is not full (has fewer than QLen elements), then not executing a MemQWr action would not prevent any responses. (Remember that a response to a write request can be issued before the value is written to memory.) We’ll return to this exception later. For simplicity, let’s require fairness for the MemQWr action too.

Our liveness condition has to assert fairness of the following actions:

\[
\text{MemQWr, MemQRd, Rsp(p), RdMiss(p), DoRd(p), DoWr(p)}
\]

for all \( p \) in Proc. We now must decide whether to assert weak or strong fairness for these actions. Weak and strong fairness are equivalent for an action that, once enabled, remains enabled until it is executed. This is the case for all of these actions except \( \text{RdMiss(p)} \) and \( \text{DoWr(p)} \). These two actions append a request to the memQ queue, and are disabled if that queue is full. A \( \text{RdMiss(q)} \) or \( \text{DoWr(q)} \) could be enabled, and then become disabled because a \( \text{RdMiss(q)} \) or \( \text{DoWr(q)} \), for a different processor \( q \), appends a request to memQ. We therefore need strong fairness for the \( \text{RdMiss(p)} \) and \( \text{DoWr(p)} \) actions. So, the fairness conditions we need are:

Weak Fairness \( \text{Rsp(p), DoRd(p), MemQWr, and MemQRd} \)

Strong Fairness \( \text{RdMiss(p) and DoWr(p)} \)

As before, let’s define \( \text{vars} \) to be the tuple of all variables.

\[
\text{vars} \triangleq \left\{ \text{memInt, mem, buf, ctl, cache, memQ} \right\}
\]
We could just write the liveness condition as
\[
(8.9) \quad \forall p \in \text{Proc} : \\WFvars(Rsp(p)) \land \WFvars(DoRd(p)) \\
\land \SFvars(RdMiss(p)) \land \SFvars(DoWr(p)) \\
\land \WFvars(MemQWr) \land \WFvars(MemQRd)
\]

However, I prefer replacing the conjunction of fairness conditions by a single fairness condition on a disjunction, as we did in Section 8.3 for the memory specification. The WF and SF Conjunction Rules (page 8.3 and 8.4) easily imply that the liveness condition (8.9) can be rewritten as
\[
(8.10) \quad \forall p \in \text{Proc} : \\WFvars(Rsp(p) \lor DoRd(p)) \\
\land \SFvars(RdMiss(p) \lor DoWr(p)) \\
\land \WFvars(MemQWr \lor MemQRd)
\]

We can now try to simplify (8.10) by applying the WF Quantifier Rule (page 93) to replace \(\forall p \in \text{Proc} : \WFvars(\ldots)\) with \(\WFvars(\exists p \in \text{Proc} : \ldots)\). However, that rule doesn’t apply; it’s possible for \(Rsp(p) \lor DoRd(p)\) to be enabled for two different processors \(p\) at the same time. In fact, the two formulas are not equivalent. Similarly, the analogous rule for strong fairness doesn’t apply. Formula (8.10) is as simple as we can make it.

Let’s return to the observation that we don’t have to execute \(\text{MemQWr}\) if the \(\text{memQ}\) queue is not full and contains only write requests. Let’s define \(QCond\) to be the assertion that \(\text{memQ}\) is not full and contains only write requests:
\[
QCond \triangleq \forall i \in 1 \ldots \text{Len}(\text{memQ}) : \text{memQ}[i][2].\text{op} = “Wr”
\]

We have to eventually execute a \(\text{MemQWr}\) action only when it’s enabled and \(QCond\) is true, which is the case iff the action \(QCond \land MemQWr\) is enabled. In this case, a \(\text{MemQWr}\) step is a \(QCond \land MemQWr\) step. Hence, it suffices to require weak fairness of the action \(QCond \land MemQWr\). We can therefore replace the second conjunct of (8.10) with
\[
\WFvars((QCond \land MemQWr) \lor MemQRd)
\]

We would do this if we wanted the specification to describe the weakest liveness condition that implements the memory specification’s liveness condition. However, if the specification were a description of an actual device, then that device would probably implement weak fairness on all \(\text{MemQWr}\) actions, so we would take (8.10) as the liveness condition.

### 8.6 Quantification

I’ve already mentioned, in Section 8.1, that the ordinary quantifiers of predicate logic can be applied to temporal formulas. For example, the meaning of the
8.6. QUANTIFICATION

For any temporal formula \( F \), the formula \( \exists r : F \) is given by

\[
\sigma \models (\exists r : F) \triangleq \exists r : (\sigma \models F)
\]

where \( \sigma \) is any behavior. The meaning of \( \forall r : F \) is similarly defined.

The symbol \( r \) in \( \exists r : F \) is usually called a bound variable. But we’ve been using the term *variable* to mean something else—something that’s declared by a variable statement in a module. The bound “variable” \( r \) is actually a constant in these formulas—a value that is the same in every state of the behavior.\(^5\) For example, the formula \( \exists r : \Box (x = r) \) asserts that \( x \) always has the same value.

The bounded quantifiers are defined in a similar way—for example,

\[
\sigma \models (\exists r \in S : F) \triangleq \exists r \in S : (\sigma \models F)
\]

For this formula to make sense, \( S \) must be a constant.\(^6\) The symbol \( r \) is declared to be a constant in formula \( F \). The expression \( S \) lies outside the scope of the declaration of \( r \), so the symbol \( r \) cannot occur in \( S \).

One can, in a similar way, define \textsc{choose} to be a temporal operator. However, it’s not needed for writing specifications, so we won’t.

We have been using the operator \( \exists \) as a hiding operator. Intuitively, \( \exists x : F \) means \( F \) with variable \( x \) hidden. In this formula, \( x \) is declared to be a variable in formula \( F \). Unlike \( \exists r : F \), which asserts the existence of a single value \( r \), the formula \( \exists x : F \) asserts the existence of a value for \( x \) in each state of a behavior.

The precise definition of \( \exists \) is a bit tricky because, as discussed in Section 8.1, the formula \( \exists x : F \) should be invariant under stuttering. To define it, we first define \( \sigma \sim_\tau \) to be true iff \( \sigma \) can be obtained from \( \tau \) (or vice-versa) by adding and/or removing stuttering steps and changing the values assigned to \( x \) by its states. To define \( \sim_\tau \) precisely, we define two behaviors \( \sigma \) and \( \tau \) to be stuttering-equivalent iff removing all stuttering steps from each of them produces the same sequence of states. Next, let \( \sigma_{x \rightarrow 0} \) be the behavior that is the same as \( \sigma \) except that, in each state, \( x \) is assigned the value 0.\(^7\) We can then define \( \sigma \sim_\tau \) to be true iff \( \sigma_{x \rightarrow 0} \) and \( \tau_{x \rightarrow 0} \) are stuttering equivalent. Finally, the meaning of \( \exists \) is defined by letting \( \sigma \models (\exists x : F) \) be true iff there exists some behavior \( \tau \) such that \( \tau \sim_\tau \sigma \) and \( \tau \models F \) are true. If you find this too confusing, don’t worry about it. For writing specifications, it suffices to just think of \( \exists x : F \) as \( F \) with \( x \) hidden.

TLA also has a temporal universal quantifier \( \forall \), defined by:

\[
\forall x : F \triangleq \neg \exists x : \neg F
\]

\(^5\)Logicians use the term *flexible variable* for a TLA variable, and the term *rigid variable* for a symbol like \( r \) that represents a constant.

\(^6\)We can let \( \exists r \in S : F \) equal \( \exists r : (r \in S) \land F \), which makes sense if \( S \) is a state function, not just a constant. However, TLA\(^+\) requires \( S \) to be a constant in \( \exists r \in S : F \). If you want it to be a state function, you have to write \( \exists r : (r \in S) \land F \).

\(^7\)The use of 0 is arbitrary; any fixed value would do.
This operator is hardly ever used.

TLA+ does not allow bounded versions of the operators  \( \exists \) and  \( \forall \). If you want to write  \( \exists x \in S : F \), you can simply write  \( \exists x : (x \in S) \wedge F \) instead.

8.7 Temporal Logic Examined

8.7.1 A Review

Let’s look at the shapes of the specifications that we’ve written so far. We started with the simple form

\[
\text{(8.11) } \text{Init} \wedge \square[\text{Next}]_{\text{vars}}
\]

where  \( \text{Init} \) is the initial predicate,  \( \text{Next} \) the next-state action, and  \( \text{vars} \) the tuple of all variables. This kind of specification is, in principle, quite straightforward.

We then introduced hiding: using  \( \exists \) to bind variables that should not appear in the specification. Those bound variables, also called hidden or internal variables, serve only to help describe how the values of the free variables (also called visible variables) change.

Hiding variables is easy enough, and it is mathematically elegant and philosophically satisfying. However, in practice, it doesn’t make much difference to a specification. A comment can also tell a reader that a variable should be regarded as internal. Explicit hiding allows implementation to mean implication. A lower-level specification that describes an implementation can be expected to imply a specification only if the specification’s internal variables, whose values don’t really matter, are explicitly hidden. Otherwise, implementation means implementation under a refinement mapping. (See Section 5.8.) However, as explained in Section 10.3, implementation often involves a refinement of the visible variables as well.

To express liveness, the specification (8.11) is strengthened to the form

\[
\text{(8.12) } \text{Init} \wedge \square[\text{Next}]_{\text{vars}} \wedge \text{Liveness}
\]

where  \( \text{Liveness} \) is the conjunction of formulas of the form  \( \text{WF}_{\text{vars}}(A) \) and/or  \( \text{SF}_{\text{vars}}(A) \), for actions  \( A \). (I’m considering universal quantification to be a form of conjunction.)

8.7.2 Machine Closure

In the specifications of the form (8.12) we’ve written so far, the actions  \( A \) whose fairness properties appear in formula  \( \text{Liveness} \) have one thing in common: they are all subactions of the next-state action  \( \text{Next} \). An action  \( A \) is a subaction of  \( \text{Next} \) if every  \( A \) step is a  \( \text{Next} \) step. Equivalently,  \( A \) is a subaction of  \( \text{Next} \) iff  \( A \)
implies Next. In almost all specifications of the form (8.12), formula Liveness should be the conjunction of weak and/or strong fairness formulas for subactions of Next. I’ll now briefly explain why.

When we look at the specification (8.12), we expect Init to constrain the initial state, Next to constrain what steps may occur, and Liveness to describe only what must eventually happen. However, consider the following formula

\[ (x = 0) \land \Box[x' = x + 1]_x \land WF_x((x > 99) \land (x' = x - 1)) \]  

(8.13)

The first two conjuncts of (8.13) assert that \( x \) is initially 0 and that any step either increments \( x \) by 1 or leaves it unchanged. Hence, they imply that if \( x \) ever exceeds 99, then it forever remains greater than 99. The weak fairness property asserts that, if this happens, then \( x \) must eventually be decremented by 1—contradicting the second conjunct. Hence, (8.13) implies that \( x \) can never exceed 99, so that formula is equivalent to

\[ (x = 0) \land \Box[(x < 99) \land (x' = x + 1)]_x \]

Conjoining the weak fairness property to the first two conjuncts of (8.13) forbids an \( x' = x + 1 \) step when \( x = 99 \).

A specification of the form (8.12) is called machine closed iff the conjunct Liveness does not constrain the initial state or what steps may occur. We almost never want to write a specification that isn’t machine closed. If we do write one, it’s almost always by mistake. Specification (8.12) is guaranteed to be machine closed if Liveness is the conjunction of weak and/or strong fairness properties for subactions of Next. This condition doesn’t apply to specification (8.13), which is not machine closed, because \((x > 99) \land (x' = x - 1)\) is not a subaction of \( x' = x + 1 \).

Liveness requirements are philosophically satisfying. A specification of the form (8.11), which specifies only a safety property, allows behaviors in which the system does nothing. Therefore, the specification is satisfied by a system that does nothing. Expressing liveness requirements with fairness properties is less satisfying. These properties are subtle and it’s easy to get them wrong. It requires some thought to determine that the liveness condition for the write-through cache, formula (8.10) on 96, does imply that every request receives a reply.

It’s tempting to express liveness properties more directly, without using fairness properties. For example, it’s easy to write a temporal formula asserting for the write-through cache that every request receives a response. When processor \( p \) issues a request, it sets \( ctl[p] \) to “rdy”. We just have to assert that a state in which \( ctl[p] = \text{"rdy"} \) is true leads to a \( Rsp(p) \) step—for every processor \( p \):

\[
(8.14) \forall p \in \text{Proc} : (ctl[p] = \text{"rdy"}) \rightsquigarrow (Rsp(p))_{vars}
\]

More precisely, this is the case for a finite or countably infinite conjunction of properties.
(The operator $\leadsto$ is defined on page 88.) While such formulas are appealing, they are dangerous. It’s very easy to make a mistake and write a specification that isn’t machine closed.

Except in unusual circumstances, you should express liveness with fairness properties for subactions of the next-state action. These are the most straightforward specifications, and hence the easiest to write and to understand. Most system specifications, even if very detailed and complicated, can be written in this straightforward manner. The exceptions are usually in the realm of subtle, high-level specifications that attempt to be very general. An example of such a specification is given in Section 11.1.

8.7.3 The Unimportance of Liveness

While philosophically important, in practice the liveness property of (8.12) is not as important as the safety part: $\text{Init} \land [\text{Next}]_{\text{vars}}$. The ultimate purpose of writing a specification is to avoid errors. Experience shows that most of the benefit from writing and using a specification comes from the safety part. On the other hand, the liveness property is usually easy enough to write. It typically constitutes less than five percent of a specification. So, you might as well write the liveness part. However, when verifying the correctness of the specification, most of your effort should be devoted to the safety part.

8.7.4 Temporal Logic Considered Confusing

The most general type of specification I’ve discussed so far has the form

$$\exists v_1, \ldots, v_n : \text{Init} \land [\text{Next}]_{\text{vars}} \land \text{Liveness}$$

where $\text{Liveness}$ is the conjunction of fairness properties of subactions of $\text{Next}$. This is a very restricted class of temporal-logic formulas. Temporal logic is quite expressive, and one can combine its operators in all sorts of ways to express a wide variety of properties. This suggests the following approach to writing a specification: express each property that the system must satisfy with a temporal formula, and then conjoin all these formulas. For example, formula (8.14) above expresses the property of the write-through cache that every request eventually receives a response.

This approach is philosophically appealing. It has just one problem: it’s practical for only the very simplest of specifications—and even for them, it seldom works well. The unbridled use of temporal logic produces formulas that are hard to understand. Conjoining several of these formulas produces a specification that is impossible to understand.

The basic form of a TLA specification is (8.15). Most specifications should have this form. We can also use this kind of specification as a building block.
Chapters 9 and Section 10.1 describe situations in which we write a specification as a conjunction of such formulas. Section 10.2 introduces an additional temporal operator $\Rightarrow$ and explains why we might want to write a specification $F \Rightarrow G$, where $F$ and $G$ have the form (8.12). But such specifications are of limited practical use. Most engineers need only know how to write specifications of the form (8.15). Indeed, they can get along quite well with specifications of the form (8.11).
Chapter 9

Real Time

This section explains how to write real-time specifications. It starts by adding real-time constraints to the hour-clock specification. As a larger example, it will describe Fischer’s algorithm or perhaps some more realistic example. It will conjecture that complex real-time specifications don’t occur in real life because it’s (a) hard to make sure that they’re correct, and (b) hard to ensure that the real-time constraints are maintained as the timing properties of the components change in later versions.
Chapter 10

Other Ways to Write Specifications

10.1 Composing Specifications

This section will explain why, in TLA, composition is conjunction. It will then describes how to write a specification as the conjunction of a system specification and an environment specification. This will be generalized to the specification of a system as the conjunction of the specifications of multiple components.

Points to be explained include:

- The distinction between interleaving representations and noninterleaving compositions.

- The composition of a set $S$ of components has the form:
  \[ \forall p \in S : Spec(p) \]

- Sometimes, a conjunct of the specification describes a “law of nature” rather than a particular component. This law of nature could actually be a restriction on how we are modeling the system. Such a restriction may be necessary if we want to compose systems specified separately to produce an interleaving representation.

- Sometimes in a composition, the state includes a variable $f$ such that $f[p]$ is maintained by component $p$. The next-state action of component $p$ might have to include a conjunct of the form $f'[p] = e$. There are two ways of handling this case:
  
  - Have the specification describe only the values of $f[p]$ for all components $p$, without describing the value of $f$. 

Add a global conjunct to the specification asserting that \( f \) is always a function whose domain is the set of components. This conjunct can be viewed as a “law of nature”.

### 10.2 Open-System Specifications

Section 10.1 describes how to write a specification as the conjunction \( M \land E \) of specifications \( M \) of a system and \( E \) of the system’s environment. Such a specification describes behaviors in which both the system and the environment operate correctly. If a behavior does not satisfy the specification, there is not formal sense in which the specification says whether it is the module or the environment that misbehaved.

An open-system specification is one in which the system behaves correctly if the environment does. One might naively write such a specification \( E \Rightarrow M \). This specification allows behaviors in which both the module and the environment misbehave. In particular, it allows behaviors in which the module does something wrong and the environment later also does something wrong. Those behaviors can’t be produced by a correct implementation of the system, since no real implementation can predict the future. While harmless in writing a single specification, they cause problems if we try to compose open-system specifications. We therefore introduce a new operator \( \Rightarrow \) and write an open-system specification in the form \( E \Rightarrow M \), a formula that is true if \( M \) either performs correctly, or else misbehaves after \( E \) has misbehaved.

The precise definition of \( \Rightarrow \) and examples of open-system specifications will be given.

### 10.3 Interface Refinement

Sections 3.1 and 3.2 give two specifications of the same asynchronous interface. In Section 5.8, we described how one spec implements the other under a refinement mapping. Here, we examine the more general type of implementation relation called interface refinement. We show that each of the asynchronous interface specifications implements the other under the suitable interface refinement. In the course of this, we see why formula \( Spec \) of module \( AsynchInterface \) doesn’t implement formula \( Spec \) of module \( Channel \) under the interface refinement that substitutes

\[
[\text{val} \mapsto \text{val}, \text{rdy} \mapsto \text{rdy}, \text{ack} \mapsto \text{ack}]
\]

for \( \text{chan} \). Under this refinement mapping, \( Spec \) implies that \( \text{chan}.\text{val}, \text{chan}.\text{rdy}, \) and \( \text{chan}.\text{ack} \) assume the correct values; but it doesn’t imply that \( \text{chan} \) is a
record containing only these three fields. We then go on to give more interesting examples of refinement mappings.
Chapter 11

Advanced Examples

11.1 Sequentially Consistent Memory

This section will provide two specifications of sequentially consistent memory, one that uses a history variable, the other that uses queues. Both specifications come from [2].

11.2 The Riemann Integral

To demonstrate how easy it is to formalize ordinary math, we give a specification of the Riemann integral—the definite integral of elementary calculus. Though sometimes written \( \int_{a}^{b} f(x) \, dx \), it’s pretty hard (though by no means impossible) to make rigorous sense of the \( dx \), so careful mathematicians usually write this integral as \( \int_{a}^{b} f \). The specification is in Figure 11.1 on the next page.
This module defines $\text{Integral}(f, a, b)$ to be the Riemann integral $\int_a^b f$ of the continuous real-valued function $f$.

To define the integral, we first define a partition $p$ of the real interval from $a$ to $b$ (or $b$ to $a$ if $a \geq b$) to be a sequence with $a = p[1] \leq \ldots \leq p[\text{Len}(p)] = b$ (or $a = p[1] \geq \ldots \geq p[\text{Len}(p)] = b$ if $a \geq b$). The diameter of a partition $p$ is the maximum of $|p[i + 1] - p[i]|$. The integral $\int_a^b f$ is the limit of

$$
\sum_{i=1}^{\text{Len}(p)-1} (p[i + 1] - p[i]) \cdot (f[p[i + 1]] + f[p[i]])/2
$$

as the diameter of the partition $p$ goes to zero.

EXTENDS Reals, Sequences

\[
\text{Abs}(r) \triangleq \begin{cases} 
\text{IF } r < 0 \text{ THEN } -r & \text{ELSE } r \\
\end{cases} \quad \text{Abs}(r) \text{ equals } |r|.
\]

\[
\text{Partition}(a, b, d) \triangleq \begin{cases} 
\text{The set of partitions of the interval from } a \text{ to } b \text{ with diameter less than } d.
\end{cases}
\]

\[
\{ p \in \text{Seq}(\text{Real}) : \begin{array}{c} 
\text{Len}[p] > 1 \\
(p[1] = a) \wedge (p[\text{Len}(p)] = b) \\
\forall i \in 1 \ldots (\text{Len}(p) - 1) : \begin{cases} 
\text{IF } a \leq b \text{ THEN } p[i] \leq p[i + 1] \\
\text{ELSE } p[i] \geq p[i + 1] 
\end{cases} \\
\wedge \text{Abs}(p[i + 1] - p[i]) < d
\end{array} \}
\]

\[
\text{PSum}(f, p) \triangleq \begin{cases} 
\text{Equality } \sum_{i=1}^{\text{Len}(p)-1} (f[p[i]] + f[p[i + 1]])/2.
\end{cases}
\]

\[
\text{LET } \text{sumf}[n \in 1 \ldots \text{Len}(p)] \triangleq \begin{cases} 
\text{Equality } \sum_{i=1}^{n} f[i].
\end{cases}
\]

\[
\text{IF } n = 1 \text{ THEN } 0 \text{ ELSE } (p[n] - p[n - 1]) \cdot ((f[p[n]] + f[p[n - 1]])/2) + \text{sumf}[n - 1] \text{ IN } \text{sumf}[\text{Len}(p)]
\]

\[
\text{Integral}(f, a, b) \triangleq \begin{cases} 
\text{Equality } \int_a^b f.
\end{cases}
\]

\[
\text{CHOOSE } r \in \text{Real} : \begin{cases} 
\forall e \in \{ s \in \text{Real} : s > 0 \} : \\
\exists d \in \{ s \in \text{Real} : s > 0 \} : \\
\forall p \in \text{Partition}(a, b, d) : \text{Abs}(r - \text{PSum}(f, p)) < e
\end{cases}
\]

Figure 11.1: A specification of the Riemann integral.
Part III

The Tools
Chapter 12

The G Parser

The G Parser is a parser for TLA\(^+\) written in Java by Jean-Charles Grégoire. The parser provides a front end for other tools, such as TLC (see Chapter 13). It can also be run by itself to find syntax errors in a specification. You should obtain directions for running the parser on your particular system when you obtain the software.

12.1 Finding an Error

When the parser reports an error, finding what caused it can be tricky. The errors that the parser detects fall into two separate classes, which are usually called syntactic and semantic errors. A syntax error is one that makes the specification grammatically incorrect, meaning that it violates the BNF grammar, or the precedence and alignment rules, described in Chapter 14. A semantic error is one that violates the legality conditions mentioned in Chapter 16. The term semantic error is misleading, because it suggests an error that makes a specification have the wrong meaning. All errors found by the parser are ones that make the specification syntactically illegal, and hence make it have no meaning at all.

The parser reads the file sequentially, starting from the beginning, and it reports a syntax error if and when it reaches a point at which it becomes impossible for any continuation to produce a grammatically correct specification. For example, if you leave out a colon and type \(\forall x \in P(x)\) instead of \(\forall x : P(x)\), the parser will print something like:

```
Encountered "P" at line 7, column 11.
Was expecting one of:
  "," ...
  <OpSymbol> ...
```
The “was expecting” list describes every possible symbol that could lead to a grammatically correct specification. Knowing what the parser was expecting can sometimes help find the error.

The parser may detect a grammatical error far from the actual mistake. For example, suppose you type [ instead of { to produce \[ x \in \ldots : P(x) \], where “…” is a very long expression. The parser will discover the error only when it sees the colon, well past the erroneous [. If you can’t find the source of an error, try the “divide and conquer” method: keep removing different parts of the module until you isolate the source of the problem.

A typical semantic error is an undefined symbol that arises because you mistype an identifier. For example, if you define an operator $Cat$ but spell it $cat$ somewhere by mistake, the parser may report

unresolved identifier cat at [line: 87, col: 6] to [87,8].

The source of a semantic error is usually easy to find.

The parser stops when it encounters the first syntactic error. It can detect multiple semantic errors in a single run.

The current version of the parser has the following limitations, which should be corrected in future versions:

- It does not detect certain semantic errors. In particular, it does not do any level checking. (See Section 16.2 on page 204.)
- It does not properly handle strings with “escape sequences” such as \\\. (See Section 15.1.8.)
- It does not properly handle parametrized instantiation. (See Section 4.2.2 on page 39.)
- It does not handle the symbol $\setminus X$. You must type $\setminus times$ to represent $\otimes$.
- It produces meaningless warnings of the form

numbers are used but NUMERAL isn’t defined

This warning is a remnant of a minor change to TLA$^+$.
- It does not handle numbers written in binary, octal, or hexadecimal notation. (See Section 15.1.9.)
Chapter 13

The TLC Model Checker

This is a description of how we expect Version 2.0 of TLC to behave. Version 1.0 of TLC does not completely implement this description; Section 13.4 on page 145 describes its limitations. Section 13.5 on page 146 describes improvements that we may make to later versions.

13.1 Introduction to TLC

TLC is a program for finding errors in TLA+ specifications. A syntactically correct specification can have two kinds of errors: it may contain “silliness” or it may not capture the intention of its author. As explained in Section 6.2, a silly expression is one whose meaning is not determined by the semantics of TLA+—for example, $3 + (1, 2)$. A specification is incorrect if whether or not some particular behavior satisfies it depends on the meaning of a silly expression. Intention isn’t a well-defined concept, and there may be a fine line between errors and unintended features.

Experience has shown that one of the most effective ways of finding both kinds of errors is by trying to verify invariance properties of a specification. TLC tries to find errors by looking for counterexamples to invariance properties—that is, to assertions of the form

\[(13.1) \text{Init} \land \Box [\text{Next}] \Rightarrow \Box \text{Inv}\]

where Inv is a state predicate. One example of an invariance property is the absence of deadlock, in which $\text{Inv}$ equals $\text{Enabled Next}$. A counterexample to this instance of (13.1) shows the possibility of deadlock—the ability of the system to reach a state in which no further progress is possible. (Of course, for some systems, deadlock may just mean successful termination.) As explained in Sections 13.3.4 and 13.3.5 below, TLC can also be used to check other properties besides invariance.
I will illustrate the use of TLC with a simple example: a specification of the alternating bit protocol for sending data over a lossy FIFO transmission line. 

The protocol might be described as a system that looks like this:

The sender receives data values from the environment over communication channel \( \text{in} \), and the receiver then sends the values to the environment on channel \( \text{out} \). The protocol uses two lossy FIFO transmission lines: the sender sends data and control information on \( \text{msgQ} \), and the receiver sends acknowledgements on \( \text{ackQ} \). The variable \( \text{lastSent} \) records the last message the sender received from the environment. The variables \( \text{sBit} \), \( \text{sAck} \), and \( \text{rBit} \) are one-bit values used to control the sending of messages between the sender and receiver.

The protocol is supposed to implement a FIFO transmission line—in fact, a bounded FIFO of length 1. Correctness of the protocol means that its specification implements (implies) formula \( \text{Spec} \) of module \( \text{BoundedFIFO} \) (Figure 4.2 on page 43), with 1 substituted for \( N \). TLC is most easily used to check invariance properties, so we want to express correctness as an invariance property.

Intuitively, correctness of the protocol means that every value sent by the environment on channel \( \text{in} \), except possibly for the last one sent, has been received by the environment on channel \( \text{out} \). To express this condition as an invariant, we add two variables, \( \text{sent} \) and \( \text{rcvd} \), that record the sequences of messages sent over channels \( \text{in} \) and \( \text{out} \), respectively. Correctness of the algorithm is then expressed in the form (13.1) with

\[
\text{Inv} \triangleq \quad \land Len(\text{rcvd}) \in \{ \text{Len(sent)} - 1, \text{Len(sent)} \} \\
\land \forall i \in 1 \ldots \text{Len(\text{rcvd})} : \text{rcvd}[i] = \text{sent}[i]
\]

Changes to the variables \( \text{sent} \) and \( \text{rcvd} \) record the communication over the channels \( \text{in} \) and \( \text{out} \). If our goal is to verify the correctness of the protocol, rather than to specify an entire system, there’s no need to represent the actual input and output channels. So, we eliminate the variables \( \text{in} \) and \( \text{out} \) from the specification. Since the last message sent by the environment is the last element of the sequence \( \text{sent} \), we don’t need the variable \( \text{lastSent} \). So, we can describe the protocol as a system that looks like this:
13.1. INTRODUCTION TO TLC

The complete protocol specification appears in module AlternatingBit in Figures 13.1 and 13.2 on the following two pages. However, for now all you need to know are the declarations:

\begin{verbatim}
CONSTANT Data   The set of data values that can be sent.
VARIABLES msgQ, ackQ, sBit, sAck, rBit, sent, rcvd
\end{verbatim}

and the types of the variables:

- \textit{msgQ} is a sequence of elements in \{0, 1\} × \textit{Data}.
- \textit{ackQ} is a sequence of elements in \{0, 1\}.
- \textit{sBit}, \textit{sAck}, and \textit{rBit} are elements of \{0, 1\}.
- \textit{sent} and \textit{rcvd} are sequences of elements in \textit{Data}.

The input to TLC consists of a TLA+ module and a configuration file. The configuration file tells TLC the names of the initial predicate, the next-state relation, and the invariant to be checked. For example, the configuration file for the alternating bit protocol will contain the declaration

\begin{verbatim}
INIT ABInit
\end{verbatim}

telling TLC to take \textit{ABInit} as the formula \textit{Init} in (13.1).

TLC works by generating behaviors that satisfy the specification. To do this for the alternating bit protocol, it needs to know what elements are in the set \textit{Data} of data values. We can tell TLC to let \textit{Data} equal the set containing two elements, named \textit{d1} and \textit{d2}, by putting the following declaration in the configuration file.

\begin{verbatim}
CONSTANT Data = \{d1, d2\}
\end{verbatim}

(We can use any sequence of letters and digits containing at least one letter as the name of an element.)

There are two ways to use TLC. The default method is to have it try to check all reachable states—that is, all states that can occur in behaviors
CHAPTER 13. THE TLC MODEL CHECKER

This specification describes a protocol for using lossy FIFO transmission lines to transmit a sequence of values from a sender to a receiver. The sender sends a data value \(d\) by sending a sequence of \((b, d)\) messages on \(msgQ\), where \(b\) is a control bit. It knows that the message has been received when it receives the ack \(b\) from the receiver on \(ackQ\). It sends the next value with a different control bit. The receiver knows that a message on \(msgQ\) contains a new value when its control bit differs from the last one it has received. The receiver keeps sending the last control bit it received on \(ackQ\).

Extends Naturals, Sequences

Constants

- **Data**: The set of data values that can be sent.

Variables

- **msgQ**: The sequence of (control bit, data value) messages in transit to the receiver.
- **ackQ**: The sequence of one-bit acknowledgments in transit to the sender.
- **sBit**: The last control bit sent by sender; it is complemented when sending a new data value.
- **sAck**: The last acknowledgment bit received by the sender.
- **rBit**: The last control bit received by the receiver.
- **sent**: The sequence of values sent by the sender.
- **rcvd**: The sequence of values received by the receiver.

**ABInit**

\[
\begin{align*}
\land \text{msgQ} = \langle \rangle \\
\land \text{ackQ} = \langle \rangle \\
\land \text{sBit} \in \{0, 1\} \\
\land \text{sAck} = \text{sBit} \\
\land \text{rBit} = \text{sBit} \\
\land \text{sent} = \langle \rangle \\
\land \text{rcvd} = \langle \rangle 
\end{align*}
\]

The initial condition:
- Both message queues are empty.
- All the bits equal 0 or 1 and are equal to each other.
- No values have been sent or received.

TypeInv

\[
\begin{align*}
\land \text{msgQ} \in \text{Seq}(\{0, 1\} \times \text{Data}) \\
\land \text{ackQ} \in \text{Seq}(\{0, 1\}) \\
\land \text{sBit} \in \{0, 1\} \\
\land \text{sAck} \in \{0, 1\} \\
\land \text{rBit} \in \{0, 1\} \\
\land \text{sent} \in \text{Seq}(\text{Data}) \\
\land \text{rcvd} \in \text{Seq}(\text{Data}) 
\end{align*}
\]

The type-correctness invariant.

**SndNewValue(d)**

\[
\begin{align*}
\land \text{sAck} = \text{sBit} & \quad \text{Enabled iff } \text{sAck equals sBit.} \\
\land \text{sent}' = \text{Append}(\text{sent}, d) & \quad \text{Append } d \text{ to } \text{sent}. \\
\land \text{sBit'} = 1 - \text{sBit} & \quad \text{Complement control bit } \text{sBit} \\
\land \text{msgQ}' = \text{Append}(\text{msgQ}, (\text{sBit'}, d)) & \quad \text{Send value on } \text{msgQ} \text{ with new control bit.} \\
\land \text{UNCHANGED } \langle \text{ackQ}, \text{sAck}, \text{rBit}, \text{rcvd} \rangle & \quad \text{Send value on } \text{msgQ} \text{ with new control bit.} 
\end{align*}
\]

Figure 13.1: The Alternating Bit Protocol (beginning)
13.1. INTRODUCTION TO TLC

\[\begin{align*}
\text{ReSndMsg} & \triangleq \text{The sender resends the last message it sent on msgQ.} \\
& \wedge \ sAck \neq sBit \\
& \wedge \ msgQ' = \text{Append}(msgQ, (sBit, sent[\text{Len}(sent)])) \\
& \wedge \ \text{UNCHANGED} \ (ackQ, sBit, sAck, rBit, sent, rcvd)
\end{align*}\]

\[\begin{align*}
\text{RcvMsg} & \triangleq \text{The receiver receives the message at the head of msgQ.} \\
& \wedge \ msgQ \neq \emptyset \\
& \wedge \ msgQ' = \text{Tail}(msgQ) \\
& \wedge \ rBit' = \text{Head}(msgQ)[1] \\
& \wedge \ \text{rcvd}' = \text{IF} \ rBit' \neq rBit \\
& \quad \text{THEN} \ \text{Append}(\text{rcvd}, \text{Head}(msgQ)[2]) \\
& \quad \text{ELSE} \ \text{rcvd} \\
& \wedge \ \text{UNCHANGED} \ (ackQ, sBit, sAck, rBit, sent)
\end{align*}\]

\[\begin{align*}
\text{SndAck} & \triangleq \wedge \ ackQ' = \text{Append}(ackQ, rBit) \\
& \wedge \ \text{UNCHANGED} \ (msgQ, sBit, sAck, rBit, sent, rcvd)
\end{align*}\]

\[\begin{align*}
\text{RcvAck} & \triangleq \wedge \ ackQ \neq \emptyset \\
& \wedge \ ackQ' = \text{Tail}(ackQ) \\
& \wedge \ sAck' = \text{Head}(ackQ) \\
& \wedge \ \text{UNCHANGED} \ (msgQ, sBit, rBit, sent, rcvd)
\end{align*}\]

\[\begin{align*}
\text{Lose}(c) & \triangleq \wedge \ c \neq \emptyset \\
& \wedge \ \exists i \in 1..\text{Len}(c): \\
& \quad c' = \begin{cases} \\
\text{IF} \ j \leq i \ \text{THEN} \ c[j] \\
\text{ELSE} \ c[j+1] \\
\end{cases} \\
& \wedge \ \text{UNCHANGED} \ (sBit, sAck, rBit, sent, rcvd)
\end{align*}\]

\[\begin{align*}
\text{LoseMsg} & \triangleq \text{Lose}(msgQ) \wedge \ \text{UNCHANGED} \ ackQ \\
\text{LoseAck} & \triangleq \text{Lose}(ackQ) \wedge \ \text{UNCHANGED} \ msgQ \\
\text{ABNext} & \triangleq \forall \ d \in \text{Data} : \text{SndNewValue}(d) \\
& \quad \vee \ \text{ReSndMsg} \vee \ \text{RcvMsg} \vee \ \text{SndAck} \vee \ \text{RcvAck} \\
& \quad \vee \ \text{LoseMsg} \vee \ \text{LoseAck}
\end{align*}\]

\[\begin{align*}
\text{vars} & \triangleq \langle \text{msgQ}, \text{ackQ}, \text{sBit}, \text{sAck}, \text{rBit}, \text{sent}, \text{rcvd} \rangle \\
\text{Spec} & \triangleq \text{ABInit} \wedge \Box[\text{ABNext}]_{\text{vars}} \\
\text{Inv} & \triangleq \wedge \ \text{Len}(\text{rcvd}) \in \{\text{Len}(\text{sent}) - 1, \text{Len}(\text{sent})\} \\
& \wedge \ \forall i \in 1..\text{Len}(\text{rcvd}) : \ \text{rcvd}[i] = \text{sent}[i] \\
\text{THEOREM} \ Spec \Rightarrow \Box[\text{Inv}]
\end{align*}\]

Figure 13.2: The Alternating Bit Protocol (end)
Figure 13.3: Module MCAlternatingBit.

EXTENDS AlternatingBit
CONSTANTS msgQLen, ackQLen, sentLen

\[ \text{SeqConstraint} \triangleq \wedge \text{Len}(\text{msgQ}) \leq \text{msgQLen} \]
\[ \wedge \text{Len}(\text{ackQ}) \leq \text{ackQLen} \]
\[ \wedge \text{Len}(\text{sent}) \leq \text{sentLen} \]

A constraint on the lengths of sequences for use by TLC.

satisfying the specification.\(^1\) You can also use TLC in simulation mode, in which it randomly generates behaviors, without trying to check all reachable states. We now consider the default mode; simulation mode is described in Section 13.2.7 on page 129.

Exhaustively checking all possible reachable states is impossible for the alternating bit protocol because the sequences of messages and values can get arbitrarily long, so there are infinitely many reachable states. To bound the number of states, we define a state predicate called the constraint that asserts bounds on the lengths of the sequences. For example, the following constraint asserts that msgQ and ackQ have lengths at most 2 and sent has length at most 3:

\[ \wedge \text{Len}(\text{msgQ}) \leq 2 \]
\[ \wedge \text{Len}(\text{ackQ}) \leq 2 \]
\[ \wedge \text{Len}(\text{sent}) \leq 3 \]

Since the sequence rcvd of values received can never be longer than the sequence sent of values sent, there is no need to constrain its length.

Instead of specifying the bounds on the lengths of sequences in this way, I prefer to make them parameters and to assign them values in the configuration file. We usually don’t want to put into the specification itself declarations and definitions that are just for TLC’s benefit. So, let’s write a new module, called MCAlternatingBit, that extends the AlternatingBit module and can be used as input to TLC. That module appears in Figure 13.3 on this page. A possible configuration file for this module appears in Figure 13.4 on the next page. The INVARIANT section tells TLC to check the invariant Inv and the type-correctness invariant TypeInv, both of which are defined in module AlternatingBit. Observe that the configuration file must specify values for all the constant parameters of the specification—in this case, the parameter Data from the AlternatingBit module and the three parameters declared in module MCAlternatingBit itself.

\(^1\)As explained in Section 2.3 (page 18), a state is an assignment of values to all possible variables. However, when discussing a particular specification, we usually consider a state to be an assignment of values to that specification’s variables. That’s what we’re doing in this chapter.
13.1. INTRODUCTION TO TLC

CONSTANTS Data = {d1, d2}
msgQLen = 2
ackQLen = 2
sentLen = 3
INIT ABInit
NEXT ABNext
INVARIANTS Inv
    TypeInv
CONSTRAINT SeqConstraint

Figure 13.4: A configuration file for module MCAlternatingBit.

The constraint and the assignment of values to the constant parameters define what we call a model of the specification. Given the specification and a model, TLC uses essentially the following algorithm to compute a set \( R \) of reachable states.

- Initialize \( R \) to the set of states that satisfy the initial predicate, and initialize the set \( U \) of unexplored states to the subset of those states that satisfy the constraint.

- While \( U \) is nonempty, do the following for each state \( s \) in \( U \). Remove \( s \) from \( U \) and compute all states \( t \) such that the step \( s \rightarrow t \) satisfies the next-state action and \( t \) is not in \( R \). For each such \( t \): add \( t \) to \( R \) and, if \( t \) satisfies the constraint, add it to \( U \).

This computes \( R \) to be the set of all states \( s \) for which there is some behavior \( \sigma \) satisfying the specification that contains \( s \) and such that every state preceding \( s \) in \( \sigma \) satisfies the constraint. TLC reports an error and stops if, during this computation, it adds to \( R \) a state that fails to satisfy the invariant.

How long it takes TLC to check a specification depends on the specification and the size of the model. Run on a 600MHz work station, TLC finds about 1000 reachable states per second for a specification as simple as that of the alternating bit protocol. For some specifications, the rate at which TLC generates states varies (inversely) with the size of the model; it can also go down as the states it generates become more complicated. For some specifications run on larger models, TLC can find fewer than one reachable state per second.

You should always begin testing a specification with a very small model, which TLC can check quickly. A small model will catch most simple errors. For example, a typical “off-by-one error” will make the next state depend upon the value of an expression \( f[e] \) when \( e \) is not in the domain of the function \( f \), and TLC will report this as an error. When a very small model reveals no more errors, you can then run TLC with larger models to try to catch more subtle errors.
One way to figure out how large a model TLC can handle is to estimate the approximate number of reachable states as a function of the parameters. For the alternating bit protocol, a calculation based on knowledge of how the protocol works shows that there are about

\[(13.2) 3D^{sentLen+1}*(msgQLen+1)*(msgQLen+2)*(ackQLen+1)*(ackQLen+2)\]

reachable states that satisfy the constraint, where \(D\) is the number of elements in \(Data\). For the model specified by the configuration file of Figure 13.4, this is about 6900 states. The estimate is somewhat high; TLC finds 2312 distinct states, 1158 of which satisfy the constraint. (Remember that TLC examines states reachable in one step from a state that satisfies the constraint.) However, what’s important is not the precise number of states, but how that number varies with the parameters. From (13.2), we see that the number of states depends only quadratically on \(msgLen\) and \(ackLen\), but exponentially on \(sentLen\). For example, letting \(Data\) have 3 instead of 2 elements and increasing \(sentLen\) from 3 to 4 can be expected to increase the number of states, and hence the running time, by a factor of 81/8, or about 10. In fact, TLC then finds 22058 states, 10050 of which satisfy the constraint.

Calculating the number of reachable states can be hard. If you can’t do it, increase the model size very gradually. The number of reachable states is typically an exponential function of the model’s parameters; and the value of \(a^b\) grows very fast with increasing values of \(b\).

Many systems have errors that will show up only on models too large for TLC to check exhaustively. After TLC has exhaustively checked your specification on as large a model as it can, you can run it in simulation mode on larger models. Simulation can’t catch all errors, but it’s worth trying.

### 13.2 How TLC Works

A program like TLC can’t handle all the specifications that can be written in a language as expressive as TLA\(^+\). To explain what kinds of specifications TLC can and cannot handle, I will now sketch how TLC works. A more complete description is given in Section 13.6.

#### 13.2.1 TLC Values

A state is an assignment of values to variables. TLA\(^+\) allows you to describe a wide variety of values—for example, the set of all sequences of prime numbers. TLC can compute only a restricted class of values. Those values are built up from the following four types of primitive values:

- **Booleans** The values `true` and `false`.


13.2. HOW TLC WORKS

**Integers**  Values like 3 and −1.

**Strings**  Values like “ab3”.

**Model Values** in the CONSTANT section of the configuration file. For example, the configuration file shown in Figure 13.4 on page 121 introduces the model values \(d_1\) and \(d_2\). Model values with different names are assumed to be different.

A TLC value is defined inductively to be either

1. a primitive value, or

2. a finite set of comparable TLC values (comparable is defined below), or

3. a function \(f\) whose domain \(D\) is a TLC value such that \(f[x]\) is a TLC value, for all \(x \in D\).

For example, the first two rules imply that

\[
(13.3) \ \{\{“a”, “b”\}, \{“b”, “c”\}, \{“c”, “d”\}\}
\]

is a TLC value because rules 1 and 2 imply that \{“a”, “b”\}, \{“b”, “c”\}, and \{“c”, “d”\} are TLC values, and the second rule then implies that (13.3) is a TLC value. Since tuples and records are functions, rule 3 implies that a record or tuple whose components are TLC values is a TLC value. For example, \(\langle 1, “a”, 2, “b” \rangle\) is a TLC value.

To complete the definition of what a TLC value is, I must explain what comparable means in rule 2. The basic idea is that two values should be comparable if the semantics of TLA\(^+\) determines whether or not they are equal. For example, strings and numbers are not comparable because the semantics of TLA\(^+\) doesn’t tell us whether or not “abc” equals 42. TLC considers a model value to be comparable to, and unequal to, any other value. The precise rules for comparability are given in Section 13.6.1.

13.2.2 How TLC Evaluates Expressions

To check whether an invariant is true in a state, TLC must evaluate the invariant, meaning that it must compute the TLC value (TRUE or FALSE) of the invariant. It does this in a straightforward way, generally evaluating subexpressions “from left to right”. In particular:

- TLC evaluates \(p \land q\) by first evaluating \(p\) and, if it equals TRUE, then evaluating \(q\).
- TLC evaluates \(p \lor q\) by first evaluating \(p\) and, if it equals FALSE, then evaluating \(q\). It evaluates \(p \Rightarrow q\) as \(\neg p \lor q\).
TLC evaluates $\text{if } p \text{ then } e_1 \text{ else } e_2$ by first evaluating $p$, then evaluating either $e_1$ or $e_2$.

TLC evaluates $\exists x \in S : p$ by enumerating the elements $s_1, \ldots, s_n$ of $S$ in some order and then evaluating $p$ with $s_i$ substituted for $x$, successively for $i = 1, \ldots, n$. TLC enumerates the elements of a set $S$ in a very straightforward way, and it gives up and declares an error if it isn't obvious from the form of the expression $S$ that the set is finite. For example, it can enumerate the elements of the following three set expressions:

$$\{0, 1, 2, 3\} \quad 0 \ldots 3 \quad \{i \in 0 \ldots 5 : i < 4\}$$

It cannot enumerate the elements of

$$\{i \in \text{Nat} : i < 4\}$$

The rules for what sets TLC can enumerate, along with a complete specification of how TLC evaluates expressions, are given elsewhere in Section 13.6.3 below.

TLC evaluates the expressions $\forall x \in S : p$ and $\text{choose } x \in S : p$ by first enumerating the elements of $S$, much the same way as it evaluates $\exists x \in S : p$. The semantics of TLA$^+$ state that $\text{choose } x \in S : p$ is an arbitrary value if there is no $x$ in $S$ for which $p$ is true. However, this case almost always arises because of a mistake, so TLC treats it as an error. Note that evaluating the expression

$$\text{if } n > 5 \text{ then } \text{choose } i \in 1 \ldots n : i > 5 \text{ else } 42$$

will not produce an error if $n \leq 5$ because TLC will not evaluate the $\text{choose}$ expression in that case.

TLC cannot evaluate “unbounded” quantifiers or $\text{choose}$ expressions—that is, expressions having one of the forms:

$$\exists x : p \quad \forall x : p \quad \text{choose } x : p$$

It cannot evaluate any expression whose value is not a TLC value; in particular, it can evaluate a set-valued expression only if it equals a finite set; it can evaluate a function-valued expression only if it equals a function with finite domain. TLC will evaluate expressions of the following forms only if it can tell syntactically that $S$ is a finite set:

$$\exists x \in S : p \quad \forall x \in S : p \quad \text{choose } x \in S : p$$

$\{x \in S : p\}$ $\{e : x \in S\}$ $[x \in S \mapsto e]$ $\text{subset } S$ $\text{union } S$

TLC can often evaluate an expression even when it can’t evaluate all subexpressions. For example, it can evaluate

$$[n \in \text{Nat} \mapsto n \ast (n + 1)][3]$$

The rules explaining exactly what TLC can evaluate appear in Section 13.6, but you don’t have to know them to use TLC.
13.2. HOW TLC WORKS

which equals the TLC value 12, even though it can’t evaluate

\[ n \in \text{Nat} \mapsto n \times (n + 1) \]

which equals a function whose domain is the set \( \text{Nat} \). (A function can be a TLC value only if its domain is a finite set.)

13.2.3 Assignment and Replacement

As we saw in the alternating bit example, the configuration file must determine the value of each constant parameter. To assign a TLC value \( v \) to a constant parameter \( c \) of the specification, we write \( c = v \) in the CONSTANT section of the configuration file. The value \( v \) may be a primitive TLC value or a finite set of primitive TLC values written in the form \( \{ v_1, \ldots, v_n \} \)—for example, \( \{1, -3, 2\} \). In \( v \), any sequence of characters like \( a1 \) or \( foo \) that is not a number, a quoted string, or \( \text{TRUE} \) or \( \text{FALSE} \) is taken to be a model value.

In the assignment \( c = v \), the symbol \( c \) need not be a constant parameter; it can also be a defined symbol. This assignment causes TLC to ignore the actual definition of \( c \) and to take \( v \) to be its value. Such an assignment is often used when TLC cannot compute the value of \( c \) from its definition. For example, TLC cannot compute the value of \( \text{NotAnS} \) from the definition:

\[ \text{NotAnS} \triangleq \text{choose } n : n \notin S \]

because it cannot evaluate the unbounded \( \text{choose} \) expression. You can override this definition by assigning \( \text{NotAnS} \) a value in the CONSTANT section of the configuration file. For example, the assignment

\[ \text{NotAnS} = \text{NS} \]

causes TLC to assign to \( \text{NotAnS} \) the model value \( \text{NS} \). TLC ignores the actual definition of \( \text{NotAnS} \). If you used the name \( \text{NotAnS} \) in the specification, you’d probably want TLC’s error messages to call it \( \text{NotAnS} \) rather than \( \text{NS} \). So, you’d probably use the assignment

\[ \text{NotAnS} = \text{NotAnS} \]

which assigns to the symbol \( \text{NotAnS} \) the model value \( \text{NotAnS} \). Remember that, in the assignment \( c = v \), the symbol \( c \) must be defined or declared in the TLA\(^+\) module, and \( v \) must be a primitive TLC value or a finite set of such values.

The CONSTANT section of the configuration file can also contain replacements of the form \( c \leftarrow d \), where \( c \) and \( d \) are symbols defined in the TLA\(^+\) module. This causes TLC to replace \( c \) by \( d \) when performing its calculations. One use of replacement is to give a value to an operator parameter. For example, suppose we wanted to use TLC to check the write-through cache specification of Section 5.6 (page 54). The WriteThroughCache module extends the MemoryInterface module, which contains the declaration

\[ \text{Note that } d \text{ is a defined symbol in the replacement } c \leftarrow d, \text{ while } v \text{ is a TLC value in the substitution } c = v. \]
CONSTANTS Send(·, ·, ·, ·), Reply(·, ·, ·, ·), · · ·

To use TLC, we have to tell it how to evaluate the operators Send and Reply. We do this by first writing a module MCWriteThroughCache that extends the WriteThroughCache module and defines two operators

\[ \text{MCSend}(p, d, \text{old}, \text{new}) \triangleq \ldots \]
\[ \text{MCReply}(p, d, \text{old}, \text{new}) \triangleq \ldots \]

We then add to the CONSTANT section of the configuration file the replacements:

\[ \text{Send} \leftarrow \text{MCSend} \]
\[ \text{Reply} \leftarrow \text{MCReply} \]

A replacement can also replace one defined symbol by another. In a specification, we usually write the simplest possible definitions. A simple definition is not always the easiest one for TLC to use. For example, suppose our specification requires an operator Sort such that Sort(S) is a sequence containing the elements of S in increasing order, if S is a finite set of numbers. Our specification in module SpecMod might use the simple definition:

\[ \text{Sort}(S) \triangleq \text{choose } s \in [1 \ldots \text{Cardinality}(S) \rightarrow S] : \]
\[ \forall i, j \in \text{DOMAIN } s : (i < j) \Rightarrow (s[i] < s[j]) \]

To evaluate Sort(S) for a set S containing n elements, TLC has to enumerate the n elements in the set [1 .. n S] of functions. This may be unacceptably slow. We could write a module MCSpecMod that extends SpecMod and defines FastSort so it equals Sort, when applied to finite sets of numbers, but can be evaluated more efficiently by TLC. We could then run TLC with a configuration file containing the replacement

\[ \text{Sort} \leftarrow \text{FastSort} \]

The following definition of FastSort requires TLC to perform only on the order of n^2 operations to sort an n-element set:

\[ \text{FastSort}(S) \triangleq \]
\[ \text{LET Insert}[s \in \text{Seq}(\text{Nat}), e \in \text{Nat}] \triangleq \]
\[ \text{IF } \text{Len}(s) = 0 \text{ THEN } \langle e \rangle \]
\[ \text{ELSE IF } e < s[1] \text{ THEN } \langle e \rangle \circ (s) \]
\[ \text{ELSE } \langle \text{Head}(s) \rangle \circ \text{Insert}[\text{Tail}(s), e] \]
\[ \text{MCS}[s \in \text{Seq}(\text{Nat}), SS \in \text{SUBSET } S] \triangleq \]
\[ \text{IF } SS = \{ \} \text{ THEN } s \]
\[ \text{ELSE LET } e \triangleq \text{choose } ee \in SS : \text{TRUE} \]
\[ \text{IN } \text{MCS}[\text{Insert}[s, e], SS \setminus e] \]

An even more efficient way to define FastSort is described in Section 13.3.3, on pages 137–138 below.
13.2.4 Overriding Modules

TLC cannot compute $2 + 2$ from the definition of $+$ contained in the standard \textit{Naturals} module. Even if we did use a definition of $+$ from which TLC could compute sums, it would not do so very quickly. Arithmetic operators like $+$ are implemented directly in Java, the language in which TLC is written. This is achieved by a general mechanism of TLC that allows a module to be overridden by a Java class that implements the operators defined in the module. When TLC encounters an \texttt{EXTENDS Naturals} statement, it reads in the Java class that overrides the \texttt{Naturals} module rather than reading the module itself. There are Java classes to override the following standard modules: \texttt{Naturals}, \texttt{Integers}, \texttt{Sequences}, \texttt{FiniteSets}, and \texttt{Bags}. (The TLC module described below in Section 13.3.3 is also overridden by a Java class.) Instructions for implementing Java classes to override other modules will appear elsewhere.

13.2.5 How TLC Computes States

When TLC evaluates the invariant, it is calculating the invariant’s value, which is either \texttt{TRUE} or \texttt{FALSE}. When TLC evaluates the initial predicate or the next-state action, it is computing a set of states—the set of initial states or the set of possible successor states (primed states) for a given (unprimed) state. I will describe how TLC does this for the next-state relation; the evaluation of the initial predicate is analogous.

Recall that a state is an assignment of values to variables. TLC begins computing the successors to a given state $s$ by assigning to all unprimed variables their values in state $s$, and assigning no values to the primed variables. It then starts computing the next-state action.

TLC computes a set of successor states by simultaneously performing a set of computations, each trying to compute a single successor state. A computation splits into multiple separate computations when multiple possibilities are encountered. TLC begins a single computation to evaluate the next-state relation. The evaluation proceeds as described in Section 13.2.2 (page 123), except that when it evaluates a subformula $A \lor B$, it splits the computation into two separate computations—in one taking the subformula to be $A$ and in the other taking the subformula to be $B$. Similarly, when it evaluates $\exists x \in S : p$, it splits the computation into multiple subcomputations, one for each element of $S$.

TLC reports an error if, in its evaluation, it encounters a primed variable that is not assigned a value—except that:

- It evaluates a conjunct of the form $x' = e$ when $x'$ has no value by evaluating $e$ and assigning to $x'$ the value it obtains.
- It evaluates a conjunct of the form $x' \in S$ as if it were $\exists v \in S : x' = v$. 

A computation that obtains the value \texttt{false} from evaluating the next-state action finds no state. A computation that obtains the value \texttt{true} finds the state determined by the values assigned to the primed variables. In the latter case, TLC reports an error if some primed variable has not been assigned a value.

Since TLC evaluates expressions from left to right, the order in which conjuncts appear can affect whether or not TLC can evaluate the next-state action. For example, it can evaluate:

\[
(x' = x + 1) \land (y' = x' + 1) \quad \text{but not} \quad (y' = x' + 1) \land (x' = x + 1)
\]

\[
(x' \in \{1, 2, 3\}) \land (x' \neq x) \quad \text{but not} \quad (x' \neq x) \land (x' \in \{1, 2, 3\})
\]

### 13.2.6 When TLC Computes What

When trying to figure out what caused an error, it helps to understand the exact order in which TLC performs its computations. TLC executes the following algorithm. This algorithm depends on whether or not TLC is checking for deadlock, which is determined by the switches with which TLC is run. (See Section 13.3.1.)

1. Precompute all constant definitions.
2. Compute all initial states.
3. For each initial state, evaluate the invariant on the state. If the invariant is satisfied, put the state in \(R\); otherwise, report an error and stop.
4. Set the queue \(U\) equal to all the initial states that satisfy the constraint, arranged in some arbitrary order.
5. If the queue \(U\) is empty, stop.
6. Remove the state \(s\) from the head of \(U\) and do the following:
   
   a. If there is some state \(t\) such that \(s \rightarrow t\) satisfies the next-state relation, then go to step 6b. If not, then if TLC is checking for deadlock, stop and report an error; otherwise go to step 5.
   
   b. Compute some states \(t\) such that \(s \rightarrow t\) satisfies the next-state relation. For each of these states \(t\) that is not in \(R\), do the following:
      
      i. If \(t\) does not satisfy the invariant, report an error and stop.
      ii. If \(t\) satisfies the constraint, add it to the end of the queue \(U\).
   
   c. If there is any other state \(t\) such that \(s \rightarrow t\) satisfies the next-state relation, then go to step 6b. Otherwise, go to step 5.

TLC can use multiple threads, so step 6 may be performed concurrently by different threads for different states \(s\).
13.2.7 Random Simulation

I have described how TLC tries to find all reachable states that satisfy a model specified by the configuration file. TLC can also be used in simulation mode to generate randomly chosen behaviors that satisfy the specification and check that they satisfy the invariant. TLC generates a behavior by randomly choosing a state $s_0$ that satisfies the initial predicate, and then randomly choosing a sequence of states $s_1, s_2, \ldots$, such that each transition $s_i \rightarrow s_{i+1}$ satisfies the next-state action. You can specify a maximum behavior length. When TLC has generated a behavior with that many states, it starts the process again to generate another behavior. TLC continues until it either finds an error or you stop it.

In simulation mode, there is no need to specify a constraint. (TLC ignores it if you do.) You will probably first run TLC on models that are small enough so it can generate all reachable states within a reasonable length of time. When you have found all the errors you can in this way, you can then search for more errors by letting TLC generate random behaviors on larger models.

For some specifications, it can take TLC a long time to generate a transition when the state gets large. For example, suppose the next-state relation contains a disjunct of the form

\[ \land \ldots \]
\[ \land \ T' \in \text{subset } S \times S \]
\[ \land \ Pred(T') \]

where $S$ and $T$ are set-valued variables and $Pred$ is some Boolean-valued operator. To choose a possible next state, TLC may have to examine all the subsets of $S \times S$ to find a value of $T'$ satisfying $Pred(T')$. If $S$ has $n$ elements, then there are $2^{n^2}$ such subsets. You will probably want to limit the length of behaviors generated by TLC to be small enough so that $S$ cannot become too large.

TLC makes its random choices using a pseudorandom number generator. Pseudorandom number generation is controlled with a seed. Running a random simulation twice with the same seed produces identical results. You can either let TLC choose a random seed, or specify the seed with a switch, as described on the next page in Section 13.3.1. (TLC always prints the seed it is using.) If TLC finds an error, you can also get it to rerun just the error trace; see the description of the arid switch on page 131.
13.3 How to Use TLC

13.3.1 Running TLC

Exactly how you run TLC depends upon what operating system you are using and how it is configured. You will probably type a command of the following form

\[ \text{program\_name switches spec\_file} \]

where:

- **program\_name** is specific to your system. It might be something like `java TLC`.
- **spec\_file** is the name of the file containing the TLA\(^+\) specification. Each TLA\(^+\) module that appears in the specification must be in a separate file named \(M.tla\), where \(M\) is the name of the module. The extension `.tla` may be omitted from `spec\_file`.
- **switches** is a possibly empty sequence of switches. TLC accepts the following switches:
  - `-config config\_file`
    Specifies that the configuration file is named `config\_file`, which must be a file with extension `.cfg`. The extension `.cfg` may be omitted from `config\_file`. If this switch is omitted, the configuration file is assumed to have the same name as `spec\_file`, except with the extension `.cfg`.
  - `-deadlock`
    Tells TLC not to check for deadlock. TLC checks for deadlock unless this switch is present.
  - `-simulate`
    Tells TLC to generate randomly chosen behaviors, instead of generating all reachable states. (See Section 13.2.7 above.)
  - `-depth num`
    This switch tells TLC that, in simulation mode, it should generate behaviors of length at most `num`. If the switch is absent, then TLC will use a default value of 100. This switch is meaningful only when the `-simulate` switch is present.
  - `-seed num`
    In simulation mode, the behaviors generated by TLC are determined by the initial “seed” given to a pseudorandom number generator. This switch tells TLC to let the seed be `num`, which must be an integer from \(-2^{63}\) to \(2^{63} - 1\). Running TLC twice in simulation mode
13.3. HOW TO USE TLC

with the same seed and aril (see the aril switch below) will produce identical results. If this switch is omitted, TLC chooses a random seed. This switch is meaningful only when the -simulate switch is present.

-aril num
The switch tells TLC that, in simulation mode, it should use num as the aril. The aril is a modifier of the initial seed. When TLC finds an error in simulation mode, it prints out both the initial seed and an aril number. Using this initial seed and aril will cause the first trace generated to be that error trace. Adding Print expressions will usually not change the order in which TLC generates traces. So, if the trace doesn’t tell you what went wrong, you can try running TLC again on just that trace to print out additional information.

-recover run_id
This switch tells TLC to start executing the specification not from the beginning, but where it left off at the last checkpoint. When TLC takes a checkpoint, it prints the run identifier. (That identifier is the same throughout an execution of TLC.) The value of run_id should be that run identifier.

-cleanup
TLC creates a number of files when it runs. When it completes, it erases all of them. If TLC finds an error, or if you stop it before it finishes, TLC can leave some large files around. The -cleanup option tells TLC to delete all files created by previous runs.

-workers num
Step 6 of the TLC execution algorithm described on page 128 can be speeded up on a multiprocessor computer by the use of multiple threads. This switch tells TLC to use num threads. There is never any point to using more threads than there are actual processors on your computer. If the switch is omitted, TLC uses a single thread.

13.3.2 Debugging a Specification

When you write a specification, it usually contains errors. The purpose of running TLC is to find as many of those errors as possible. Hopefully, an error in the specification causes TLC to report an error. The challenge of debugging is to find the error in the specification that leads to the error that TLC reports. Before addressing that problem, let’s first understand TLC’s output when it finds no error.


TLC’s Normal Output

When you run TLC, the first thing it prints is the version number and creation date—something like:

TLC Version 1.0 of 26 May 1999

Always include this information when reporting any problems with TLC. Next, TLC describes the mode in which it’s being run. The possibilities are

Model-checking

in which it is exhaustively checking all reachable states, or

Running Random Simulation with seed 1901803014088851111.

in which it is doing random simulation using the indicated seed. (See section 13.2.7.) Let’s suppose its doing model checking. TLC next types something like:

Finished computing initial states:
4 states generated, with 2 of them distinct.

This indicates that, when evaluating the initial predicate, TLC generated 4 states, among which there were 2 distinct ones. TLC then types one or more messages like

Progress: 2846 states generated, 984 distinct states found.
856 states left on queue.

This message indicates that TLC has thus far generated and examined 2846 states, it has found 984 distinct ones, and the queue $U$ of unexplored states contains 856 states. (See Section 13.2.6 on page 128.) After running for a while, TLC generates these progress reports about once every five minutes. For most specifications, the number of states on the queue increases monotonically at the beginning of the execution and decreases monotonically at the end. The progress reports therefore provide a useful guide to how much longer the execution is likely to take.

When TLC successfully completes, it prints

Model checking completed. No error has been found.

It then prints something like:

Estimates of the probability that TLC did not check all reachable states because two distinct states had the same fingerprint:
    calculated (optimistic): .000003
    based on the actual fingerprints: .00007
As explained in Section 13.6.4 on page 157, there is a chance that TLC did not examine the complete set of reachable states. TLC prints two different estimates of that probability. The first estimate is generally lower and more optimistic; the second is perhaps a more realistic one.

Finally, TLC prints something like

```
2846 transitions taken. 984 states discovered. 0 states left on queue.
```

with the grand totals.

While TLC is running, it may also print something like

```
-- Checkpointing run states/99-05-20-15-47-55 completed
```

This indicates that it has written a checkpoint that you can use to restart TLC in the event of a computer failure. (As explained in Section 13.3.6 on pages 143–144, checkpoints have other uses as well.) The run identifier

```
```

is used with the `-recover` switch to restart TLC from where the checkpoint was taken. (If only part of this message was printed—for example, because your computer crashed while TLC was taking the checkpoint—there is an extremely small chance that all the checkpoints are corrupted and you must start TLC again from the beginning.)

**Error Reports**

The first problems you find in your specification will probably be syntax errors. TLC reports them with

```
ParseException in parseSpec:
```

followed by the error message generated by the parser. Chapter 12 explains how to interpret the parser’s error messages. (Note: TLC Version 1.0 does not use the parser’s full error detection mechanism; you should check your specification with the parser before running TLC on it.)

After parsing, TLC executes two basic phases: in the first, it computes the initial states and in the second it generates the successor states of states on the queue of unexplored states. You can tell which phase TLC is in by whether or not it has printed the “initial states computed” message.

TLC evaluates the invariant in both phases—on the initial states in the first phase, and on newly generated successor states in the second. TLC’s most straightforward error report occurs when the invariant is violated. Suppose we introduce an error into our alternating bit specification (Figures 13.1 and 13.2
on pages 118 and 119) by replacing the first conjunct of the invariant \( \text{TypeInv} \) with

\[
\land \text{msgQ} \in \text{Seq(Data)}
\]

TLC quickly finds the error and types

**Invariant TypeInv is violated**

It next prints a minimal-length\(^2\) behavior that leads to the state not satisfying the invariant:

The behavior up to this point is:

**STATE 1:**

\[
\land \text{rBit} = 0 \\
\land \text{ackQ} = << >> \\
\land \text{rcvd} = << >> \\
\land \text{sent} = << >> \\
\land \text{sAck} = 0 \\
\land \text{sBit} = 0 \\
\land \text{msgQ} = << >>
\]

**STATE 2:**

\[
\land \text{rBit} = 0 \\
\land \text{ackQ} = << >> \\
\land \text{rcvd} = << >> \\
\land \text{sent} = << d1 >> \\
\land \text{sAck} = 0 \\
\land \text{sBit} = 1 \\
\land \text{msgQ} = << << 1, d1 >> >>
\]

Observe that TLC prints each state as a TLA\(^+\) predicate that determines the state. When printing a state, TLC describes functions using the operators :> and @@, where

\[(d_1 :> e_1 @@ \ldots d_n :> e_n)\]

is the function \( f \) with domain \( \{d_1, \ldots, d_n\} \) such that \( f[d_i] = e_i \), for \( i = 1, \ldots, n \).

These operators are defined by the \( TLC \) module, described in Section 13.3.3 (page 13.3.3). For example, the sequence \( \langle \text{“ab”}, \text{“cd”} \rangle \), which is a function with domain \( \{1, 2\} \), can be written as

\[(1 :> \text{“ab”} @@ 2 :> \text{“cd”})\]

---

\(^2\)When using multiple threads, it there is a slight chance that there exists a shorter behavior that also violates the invariant.
TLC generally prints values the way they appear in the specification, so a sequence will be printed as a sequence, rather than with this function notation.

The hardest errors to locate are usually the ones that TLC detects when evaluating an expression. They may occur when evaluating the initial predicate, the next-state action, or an invariant. These errors are detected when TLC is forced to evaluate an expression that it can’t handle, or one that is “silly” because its value is not specified by the semantics of TLA	extsuperscript{+}. As an example of a silliness error, let’s return again to the alternating bit protocol and replace the then clause in the fourth conjunct of the definition of \textit{RcvMsg} with

\begin{verbatim}
THEN Append(rcvd, Head(msgQ)[3])
\end{verbatim}

TLC discovers the error because the elements of \textit{msgQ} are pairs, so 3 is not an element in the domain of \textit{Head(msgQ)}. TLC reports the error by printing:

\begin{quote}
Error: Applying tuple to an integer out of domain.
\end{quote}

It then prints a behavior leading to the error. TLC finds the error when evaluating the next-state action to compute the successor states for some state \textit{s}, and \textit{s} is the last state in that behavior. Had it found the error when evaluating the invariant, the behavior would end with the state in which TLC was evaluating the invariant. Finally, TLC prints the location of the error:

\begin{quote}
The error occurred when TLC was evaluating the nested expressions at the following positions:
0. Line 44, column 6 to line 45, column 61 in \texttt{AlternatingBit}
\end{quote}

This position identifies the entire conjunct

\begin{verbatim}
rcvd' = IF rBit' ≠ rBit THEN Append(rcvd, Head(msgQ)[3]) ELSE rcvd
\end{verbatim}

TLC is not very precise when indicating the position of an error; usually it just narrows it down to a conjunct or disjunct of a formula. In general, it prints a tree of nested expressions—higher-level ones first. This can be helpful if the error occurs when evaluating the definition of an operator that is used in several places.

\section*{Debugging}

Tracking down an error can be difficult—especially in TLC Version 1, which does not provide you with very much information. (Sometimes, it doesn’t even print the location of the error.) The \textit{TLC} module provides some operators that can help in debugging.\textsuperscript{3}

\textsuperscript{3}Actually, it is the Java class that overrides the \textit{TLC} module that provides the useful functionality of these operators.
The TLC module defines the operator Print so that Print(out, val) equals val. But, when TLC evaluates this expression, it prints the values of out and val. You can add Print expressions to a specification to help locate an error. For example, if your specification contains

\[
\begin{align*}
&\land \text{Print(“a”, TRUE)} \\
&\land P \\
&\land \text{Print(“b”, TRUE)}
\end{align*}
\]

and TLC prints the "a" but not the "b" before reporting an error, then the error occurs while TLC is evaluating \( P \). If you know where the error is but don’t know why it’s occurring, you can add Print expressions to give you more information about what values TLC has computed.

To understand what will be printed when, you must know how TLC evaluates expressions, which is explained in Section 13.2.2 on page 123. An expression is typically evaluated many times by TLC, so inserting a Print expression in the specification can produce a lot of output. You can limit the amount of output by putting the Print expression inside an if/then expression, so it is executed only in interesting cases.

Two other debugging operators defined in the TLC module are Assert and JavaTime. The expression Assert(val, out) equals val; but if val does not equal TRUE, then evaluating this expression causes TLC to print the value of out and to halt—just as if it had found an error. The expression JavaTime equals the current time at which TLC evaluates the expression. That time is given as the number of milliseconds elapsed since 00:00 Universal Time on 1 January 1970, modulo 2^{31}. If TLC is generating states slowly, using the JavaTime operator in conjunction with Print expressions can help you understand why. If TLC is spending too much time evaluating an operator, you may be able to replace the operator with an equivalent one that TLC can evaluate more efficiently. (See Section 13.2.3 on page 125.)

### 13.3.3 The TLC Module

The standard TLC module, in Figure 13.5 on the next page, contains operators that are handy when using TLC. The module on which you run TLC will normally extend the TLC module. The TLC module is overridden by its Java implementation.

Module TLC first defines three operators Print, Assert, and JavaTime that are of no use except when running TLC. They are explained in Section 13.3.2 on the preceding page. That section also describes the operators \( : > \) and \( @ @ @ \), which are used for explicitly writing functions. These operators could be useful when writing specifications, even if you’re not using TLC.

The module next defines BoundedSeq\( (S, n) \) to be the set of sequences of elements of \( S \) of length at most \( n \). TLC cannot evaluate BoundedSeq\( (S, n) \) from
13.3. HOW TO USE TLC

MODULE TLC

OPERATORS FOR DEBUGGING

Print(out, val) \triangleq val  
Causes TLC to print the values out and val.

Assert(val, out) \triangleq val  
Causes TLC to report an error and print out if val is not true.

JavaTime \triangleq \text{CHOOSE } n : n \in \mathbb{Nat}  
Causes TLC to print the current time, in milliseconds elapsed since 00:00 on 1 Jan 1970 UT, modulo $2^{31}$.

OPERATORS FOR REPRESENTING FUNCTIONS

\[
d : \! \! > \! \! e \triangleq [x \in \{d\} \mapsto e]
\]

\[
f @@ g \triangleq \begin{cases} 
{x \in (\text{DOMAIN } f) \cup (\text{DOMAIN } g) \mapsto} \\
\text{if } x \in \text{DOMAIN } f \text{ then } f[x] \text{ else } g[x]
\end{cases}
\]

The function \( f \) with domain \( \{d_1, \ldots, d_n\} \) such that \( f[d_i] = e_i \), for \( i = 1, \ldots, n \) can be written:

\[
(d_1 : > e_1 @@ \ldots @ d_n : > e_n)
\]

OPERATORS FOR MODIFYING A SPECIFICATION

LOCAL INSTANCE Naturals,Sequences  
The keyword \text{local} means that definitions from the instantiated modules are not obtained by a module that extends TLC.

BoundedSeq(S, n) \triangleq \{s \in Seq(S) : \text{Len}(s) \leq n\}  
Sequences of length at most \( n \) having elements in \( S \).

SortSeq(s, \prec) \triangleq The result of sorting sequence \( s \) according to the ordering \( \prec \).

let Perm \triangleq \text{CHOOSE } p \in [1 \ldots \text{Len}(s) \mapsto 1 \ldots \text{Len}(s)]:
\begin{align*}
\forall i, j \in [1 \ldots \text{Len}(s) : i < j & \Rightarrow \land p[i] \neq p[j] \\
\land \lor s[p[i]] & \prec s[p[j]] \\
\lor s[p[i]] = s[p[j]]
\end{align*}

\]

\[
\text{in } [i \in [1 \ldots \text{Len}(s) \mapsto s[\text{Perm}[i]]]
\]

FApply(f, Op(\prec, \prec), \text{Identity}) \triangleq This defines FApply(f, +, 0) to equal \( \sum_{i \in \text{DOMAIN } f} f[i] \).

let FA[S \in \text{SUBSET DOMAIN } f] \triangleq \begin{cases} 
\text{Identity} \\
\text{else let } s \triangleq \text{CHOOSE } s \in S : \text{true}
\end{cases}
\text{in } \text{Op}(f[s], FA[S \setminus \{s\}])
\text{in } FA[\text{DOMAIN } f]
\]

Figure 13.5: The standard module TLC

the definition in the module, since it would first have to evaluate the infinite set Seq(n). TLC could evaluate it from the equivalent definition

\[
\text{BoundedSeq}(S, n) \triangleq \text{UNION } \{[1 \ldots m \mapsto S] : m \in 0 \ldots n\}
\]

but evaluation is faster with the overriding Java implementation.

The TLC module next defines the operator \text{SortSeq}. If \( s \) is a finite sequence and \( \prec \) is a total ordering relation on its elements, then \text{SortSeq}(s, \prec) \) is the
sequence obtained from \( s \) by sorting its elements according to \(<\). For example, 
\[
\text{SortSeq}((3,1,3,8), >) = (8,3,3,1).
\]
The Java implementation of \text{SortSeq} allows TLC to evaluate it more efficiently than a user-defined sorting operator.

For example, because the following definition of the operator \text{FastSort}, which makes use of \text{SortSeq} to do the actual sorting, TLC can evaluate it faster than the definition given above on page 126.

\[
\text{FastSort}(S) \triangleq \\
\text{LET } \text{MakeSeq}[SS \in \text{BoundedSubSet}(S, \text{Cardinality}(S))] \triangleq \\
\quad \text{if } SS = \emptyset \text{ THEN } \emptyset \\
\quad \text{ELSE LET } ss \triangleq \text{choose } ss \in SS : \text{true} \\
\quad \text{IN } \text{SortSeq}(\text{MakeSeq}[S], <)
\]

The TLC module ends by defining the operator \text{FApply}. If \( f \) is a function with finite domain, and \( Op \) is an operator that takes two arguments, then \( \text{FApply}(f, Op, \text{Id}) \) equals

\[
\text{Op}(f[d_1], \text{Op}(f[d_2], \text{Op}(\ldots \text{Op}(f[d_n], \text{Id})\ldots))
\]

where \( d_1, \ldots, d_n \) are the elements in the domain of \( f \), listed in some arbitrary order. Using \text{FApply} to avoid a recursive definition can speed up TLC. For example, the factorial function \text{fact} can be defined by

\[
\text{fact}(n) \triangleq \text{FApply}([i \in 1 \ldots n \mapsto i], *, 1)
\]

TLC can compute factorials from this definition faster than from the standard recursive definition on page 54.

### 13.3.4 Checking Action Invariance

Section 5.7 on 60 discusses two different kinds of invariants. A state predicate \( \text{Inv} \) that satisfies \( \text{Spec} \Rightarrow \Box \text{Inv} \) is called an invariant of the specification \( \text{Spec} \). This is the kind of invariance property that TLC checks. If \( \text{Inv} \) satisfies \( \text{Inv} \land [\text{Next}]_{\mu} \Rightarrow \text{Inv}' \), then \( \text{Inv} \) is called an invariant of the action \( [\text{Next}]_{\mu} \). It’s not hard to see that \( \text{Inv} \) is an invariant of action \( [\text{Next}]_{\mu} \) iff it satisfies \( \text{Inv} \land \Box [\text{Next}]_{\mu} \Rightarrow \Box \text{Inv} \).

We can therefore check if \( \text{Inv} \) is an invariant of an action \( \text{Next} \) by running TLC with \( \text{Inv} \) as both the initial condition and the invariant.

As an example, let’s return to the protocol of module \text{AlternatingBit} (pages 118–119). The predicate \( \text{Inv} \) is an invariant of the specification \( \text{Spec} \), but not of the next-state action \( \text{ABNext} \). To prove that next is an invariant of \( \text{Spec} \), we must find an invariant of \( \text{ABNext} \) that is true in an initial state and implies \( \text{Inv} \). Such an invariant \( \text{ABInv} \) is defined in Figure 13.6 on the next page. Don’t worry about the details of this invariant; just observe that it is the conjunction

When \( v \) is the tuple of all relevant variables, \( \text{Inv} \) is an invariant of \( [\text{Next}]_{\mu} \) iff it is an invariant of \( \text{Next} \).
\( ABInv \triangleq \)
\[ \land TypeInv \land Inv \land IF \ rBit = sBit \]
\[ THEN \land len(sent) = len(rcvd) \land (sent = \{\}) \Rightarrow (msgQ = \{\}) \land (sAck = sBit) \]
\[ \land \forall i \in 1 \ldots len(msgQ) : msgQ[i] = \langle sBit, sent[Len(sent)] \rangle \]
\[ ELSE \land len(sent) \neq len(rcvd) \land (msgQ \neq \{\}) \Rightarrow \exists i \in 0 \ldots len(msgQ) : \]
\[ \land (i \neq 0) \Rightarrow (rcvd \neq \{\}) \land \forall j \in 1 \ldots i : msgQ[j] = \langle rBit, rcvd[len(rcvd)] \rangle \land \forall j \in (i + 1) \ldots len(msgQ) : \]
\[ msgQ[j] = \langle sBit, sent[Len(sent)] \rangle \]
\[ \land sAck \neq sBit \land IF \ rBit = sAck \]
\[ THEN \forall i \in 1 \ldots len(ackQ) : ackQ[i] = rBit \]
\[ ELSE (ackQ \neq \{\}) \Rightarrow \exists i \in 0 \ldots len(ackQ) : \forall j \in 1 \ldots i : ackQ[j] = sAck \land \forall j \in (i + 1) \ldots len(ackQ) : ackQ[j] = rBit \]

Figure 13.6: An invariant of the alternating bit protocol’s next-state action.

of the type invariant TypeInv, the invariant Inv of the specification, and another formula.

Before checking that \( ABInv \) is an invariant of the next-state relation, we should first make sure that it’s an invariant of the specification. We add the definition of \( ABInv \) to module MCAlternatingBit, modify the configuration file to use \( ABInv \) as the invariant, and run TLC. After correcting the inevitable typing mistakes, we find that \( ABInv \) does appear to be an invariant of the specification. (Remember that TLC only checks \( ABInv \) on a finite model, it doesn’t prove invariance.) We now try running TLC again letting \( ABInv \) also be the initial predicate. TLC reports the error:

While computing initial states, TLC was trying to compute the values of a variable \( v \) from an expression \( v \in S \), but \( S \) was not enumerable.

Recall that TLC computes the initial states by computing the conjuncts of the initial predicate in order. (See Section 13.2.5 on page 127). From the definition of TypeInv, we see that it first tries to compute the possible values of msgQ by evaluating

\[ msgQ \in Seq(\{0, 1\} \times Data) \]
Since \( \text{Seq}(\{0,1\} \times \text{Data}) \) is an infinite set, containing sequences of any length, this yields an infinite number of possible values of \( \text{msgQ} \), so TLC gives up. (Remember that TLC first computes all states satisfying the initial condition, then it throws away those that don’t satisfy the constraint.) So, we must modify \( \text{ABInv} \) to specify only a finite set of initial values. We add to the \( \text{MCA}lternatingBit \) module a predicate \( \text{BTypeInv} \) that is a bounded version of \( \text{TypeInv} \). For example, it specifies the initial value of \( \text{msgQ} \) by

\[
\text{msgQ} \in \text{BoundedSeq}(\{0,1\} \times \text{Data}, \text{msgQLen})
\]

where the operator \( \text{BoundedSeq} \), defined in the TLC module, is explained on pages 136–137, and \( \text{msgQLen} \) is the parameter of \( \text{MC}lternatingBit \) used in the constraint to bound the length of \( \text{msgQ} \). We then define \( \text{BdedABInv} \) to be the same as \( \text{ABInv} \), except with \( \text{TypeInv} \) replaced by \( \text{BTypeInv} \), and run TLC with \( \text{BdedABInv} \) as the initial condition and \( \text{ABInv} \) as the invariant, using the same constraint as before. (Do you see why we can’t use \( \text{BdedABInv} \) as the invariant? If not, review Section 13.2.6.)

When computing the initial states, TLC examines every state satisfying \( \text{BTypeInv} \), throwing away those that don’t satisfy the rest of \( \text{BdedABInv} \). Since \( \text{BoundedSeq}(S,n) \) has more than \( \text{Cardinality}(S)^n \) elements, the number of initial states that TLC examines is greater than

\[
8 \cdot 2^{\text{msgQLen} + \text{ackQLen}} \cdot D^{\text{msgQLen} + 2 \cdot \text{sentLen}}
\]

where \( D \) is the cardinality of \( \text{Data} \). (I am taking \( \text{sentLen} \) to be the bound on the length of the queues \( \text{sent} \) and \( \text{rcvd} \).) This number grows much faster than the number of reachable states, estimated by formula (13.2) on page 122. Typically, TLC can check an invariant of an action only on a very small model—much smaller than the models on which it can check an invariant of a specification.

We can use \( \text{BdedABInv} \) as the initial predicate only because its first conjunct is the type invariant \( \text{BTypeInv} \). If we interchange its first two conjuncts, then TLC will try to evaluate \( \text{Inv} \) before evaluating \( \text{BTypeInv} \). Evaluating the first conjunct

\[
\text{Len}(\text{rcvd}) \in \{\text{Len}(\text{sent}) - 1, \text{Len}(\text{sent})\}
\]

of \( \text{Inv} \) causes TLC to produce the error message

The identifier \text{rcvd} is not assigned a value in the current context...

When using TLC to check that a predicate \( P \) is an invariant of an action, evaluating \( P \) must specify the value of each variable \( v \) with a conjunct of the form \( v \in S \) before that value is used.
13.3.5 Checking Step Simulation

In industrial applications, a TLA+ specification is likely to be the only formal description of a system. Such a specification is correct if it means what we intended, which is not a formalizable concept. By formalizing some of our intentions as invariants, we can use TLC to help catch errors in the specification. Sometimes, a specification $S_L$ is a lower-level view of a system for which we have a higher-level specification $S_H$. We usually regard $S_H$ as the system’s specification and $S_L$ as an implementation, and take correctness of $S_L$ to mean that it implies $S_H$. This situation occurred in Chapter 5, with the implementation $S_L$ being the specification of the write-through cache (formula $Spec$ of module $WriteThroughCache$) and the specification $S_H$ being the specification of a linearizable memory (formula $Spec$ of the $Memory$ module).

We saw in Section 5.8) that this requires proving a formula of the form:

$$Init \land \square [Next] \Rightarrow HInit \land \square [HNext]_{hv}$$

This formula is true iff the following two conditions hold:

1. $Init$ implies $HInit$

2. In every behavior satisfying $Init \land \square [Next]$, every step is an $[HNext]_{hv}$.

As explained on page 136, the TLC module defines a special operator $Assert$ with the property that evaluating $Assert(P, msg)$ causes TLC to halt and print the value $msg$ if $P$ is false. Since TLC evaluates expressions from left to right, it will check condition 1 if run with the initial predicate

$$Init \land Assert(HInit, \text{“HInit false”})$$

and it will check condition 2 if run with that initial predicate and the next-state action

$$Next \land Assert([HNext]_{hv}, \text{“HNext false”})$$

As an example, let’s return again to the alternating bit protocol. Module $ABSpec$ of Figure 13.7 on the next page contains a high-level specification $HSpec$ of the transmission protocol that is implemented by the alternating bit protocol. (In that specification, values are simply transferred directly from $sent$ to $rcvd$.)

We check that the alternating bit protocol implies specification $HSpec$ by running TLC on module $MCABSpec$ in Figure 13.8 on the next page, using the initial predicate $MCInit$, the next-state relation $MCNext$, the constraint $SeqConstraint$, and no invariant. Note that module $MCABSpec$ can extend both the $ABSpec$ and $AlternatingBit$ modules because there happen to be no name conflicts between the two modules; usually we would have to instantiate one of the modules with renaming.
CHAPTER 13. THE TLC MODEL CHECKER

EXTENDS Sequences

CONSTANT Data

VARIABLES sent, rcvd

Init $\triangleq$ \( \land sent = \langle \rangle \)
\( \land rcvd = \langle \rangle \)

Send\((d)\) $\triangleq$ \( \land sent = rcvd \)
\( \land sent' = \text{Append}(sent, d) \)
\( \land \text{UNCHANGED} \ rcvd \)

Rcv $\triangleq$ \( \land sent \neq rcvd \)
\( \land rcvd' = \text{Append}(rcvd, sent[\text{Len}(sent)]) \)
\( \land \text{UNCHANGED} \ sent \)

Next $\triangleq$ Rcv \( \lor (\exists \ d \in \text{Data} : \text{Send}(d)) \)

HSpec $\triangleq$ Init \( \land \square[\text{Next}](sent, rcvd) \land \text{WF}(sent, rcvd)(\text{Next}) \)

Figure 13.7: A higher-level specification of the alternating bit protocol.

13.3.6 Some Hints

Here are some suggestions for using TLC effectively.

Start small

The specification of a real system probably has more reachable states than you expect. When you start testing it with TLC, use the smallest model you possibly can. Let every set parameter, such as a set of processes, have only one element. Let queues be of length one—or even of length zero, if possible. A specification that has not been tested probably has lots of trivial errors that can be found

EXTENDS AlternatingBit, ABSpec

CONSTANTS msgQLen, ackQLen, sentLen

MCInit $\triangleq$ ABInit \( \land \text{Assert}(\text{Init, “Init False”}) \)

MCNext $\triangleq$ ABNext \( \land \text{Assert}([\text{Next}](sent, rcvd), “\text{Next False”}) \)

SeqConstraint $\triangleq$ \( \land \text{Len}(msgQ) \leq \text{msgQLen} \)
\( \land \text{Len}(ackQ) \leq \text{ackQLen} \)
\( \land \text{Len}(sent) \leq \text{sentLen} \)

Figure 13.8: A module for checking the alternating bit protocol.
with any model. You will find them faster with a tiny model.

Be suspicious of success

The easiest way to build a system that satisfies an invariant is to have it do nothing. If TLC finds no states in which the invariant is false, that may be because it isn’t generating many states. TLA+ specifications use liveness properties to rule out that possibility. TLC doesn’t check liveness properties, so you’ll have to use some tricks to look for errors that affect liveness. One trick is to check that progress is possible—namely, that TLC reaches the states that you expect it to. This can be done by using an invariant asserting that the expected states are not reached, and making sure that TLC reports that the invariant is violated. For example, we can check that the alternating bit protocol allows progress by using an invariant asserting that the length of the \textit{rcvd} queue is less than \textit{sentLen}. You can also check that states are reached by the judicious use of \textit{Print} or \textit{Assert} expressions. (See Section 13.3.2.)

Make model values parameters

When debugging a specification, you may want to refer to model values within TLA+ expressions. You can do this by making the model values constant parameters. For example, we might add to module \textit{MCAlternatingBit} the declaration

\[
\text{CONSTANTS } d_1, d_2
\]

and add to the \texttt{CONSTANT} section of the configuration file the assignments

\[
d_1 = d_1 \quad d_2 = d_2
\]

which assign to the constant parameters \(d_1\) and \(d_2\) the model values \(d_1\) and \(d_2\), respectively.

Don’t start over after every error

After you’ve eliminated the errors that are easy to find, TLC may have to run for a long time before finding an error. It often takes more than one try to correctly correct an error, and it can be frustrating to run TLC for an hour only to find that you made a silly mistake in the correction. If the error was discovered when taking a step from a correct state, then it’s a good idea to check your correction by starting TLC from that state. You can do this by using the state printed by TLC to define the initial predicate to be used by TLC.

Another way to avoid starting from scratch after an error is by using checkpoints. A checkpoint saves the set of reached states and the queue of unexplored states.\footnote{Some of the reached states may not be saved, so TLC may explore again some states that had been reached before the checkpoint.} It does not save any other information about the specification. You can
restart a specification from a checkpoint even if you have changed the specification, as long as the specification's variables and the values that they can assume haven't changed. If TLC finds an error after running for a long time, you may want to continue it from the last checkpoint instead of having it recheck all the states it had already checked.

**Check everything you can**

A single invariant seldom expresses correctness of a specification. Write and let TLC check as many different invariance properties as you can. If you think that some predicate should be an invariant, let TLC test if it is. Discovering that the predicate is not an invariant may not signal an error, in the specification, but it will probably teach you something about your specification.

**Try to check liveness**

Although TLC can't directly check liveness properties, you can sometimes check them indirectly by modifying the specification. A liveness property asserts that something must eventually happen. Sometimes, you can check such a property by checking that the property must hold after the system has taken a certain number of steps. For example, suppose you want to check that the specification

\[
\text{Spec} = \text{Init} \land \Box \text{[Next]} \land \text{WF} \text{(Next)}
\]

satisfies the property \( P \leadsto Q \) for some predicates \( P \) and \( Q \). In most cases, \( \text{Spec} \) implies \( P \leadsto Q \) if it implies that, whenever \( P \) is true, \( Q \) must become true within \( N \) steps, where \( N \) is some function of the constant parameters. You can check this by adding a variable \( ctr \) that counts the number of steps taken since \( P \) became true, and is reset when \( Q \) becomes true. You can check that \( P \) always leads to \( Q \) within \( N \) steps by checking that \( ctr \) is never greater than \( N \). More precisely, you can define a new specification whose initial predicate is

\[
\text{Init} \land (ctr = \text{IF } P \land \neg Q \text{ THEN } 0 \text{ ELSE } -1)
\]

and next-state action is

\[
\land \text{Next} \\
\land ctr' = \text{IF } \neg Q' \land (ctr \geq 0 \lor P') \text{ THEN } ctr + 1 \text{ ELSE } -1
\]

You can then use TLC to check that \( ctr \leq N \) is an invariant of this specification.

**Use symmetry to save time**

For many specifications, TLC will take a long time to check all the reachable states for a nontrivial model. One way to have it use less time is to reduce the number of reachable states it must check. There is often some symmetry in the
specification that makes certain states equivalent in terms of finding errors. Two
states \( s \) and \( t \) are equivalent if a behavior starting from state \( s \) can produce an
error if one starting from state \( t \) can. If \( s \) and \( t \) are equivalent, there is no need
to explore states reachable from both of them. For example, the alternating bit
protocol does not depend on the actual values being sent. So, permuting all the
data values in any state produces an equivalent state.

The constraint can often be used to keep TLC from exploring the successors
of different equivalent states. For example, we can exploit the symmetry of the
alternating bit protocol under permutation of data values as follows. We use
a model in which \( Data \) is a set of numbers and conjoin to the constraint the
requirement that the elements of \( sent \) are sorted:

\[
\forall i, j \in 1 \ldots \text{Len}(\text{sent}) : (i \leq j) \Rightarrow (\text{sent}[i] \leq \text{sent}[j])
\]

For the model size specified by the configuration file of Figure 13.4, this extra
constraint reduces the number of reachable states from 2312 to 1482. When the
number of elements in \( Data \) is increased to 3 and \( \text{sentLen} \) is increased to 4, it
reduces the number of reachable states from 22058 to 6234.

13.4 What TLC Doesn’t Do

We would like TLC to generate all the behaviors that satisfy a specification. But
no program can do this for an arbitrary specification. I have already mentioned
various limitations of TLC. There are some other limitations that you may
stumble on. One of them is that the Java classes that override the \textit{Naturals} and
\textit{Integers} modules will handle only numbers in the interval \((-2^{31} \ldots 2^{31} - 1)\).

An easier goal would be for every behavior that TLC does generate to sat-
ify the (safety part) of the specification. However, for reasons of efficiency,
TLC doesn’t always meet this goal. In particular, it doesn’t preserve the pre-
cise semantics of \textit{choose}. As explained in Section 15.1, if \( S \) equals \( T \), then
\textit{choose} \( x \in S : P \) should equal \textit{choose} \( x \in T : P \). However, if the expressions
don’t have a uniquely-defined value, then TLC guarantees this only if \( S \) and \( T \)
are syntactically the same. For example, TLC might compute different values
for the two expressions

\[
\text{choose } x \in \{1, 2, 3\} : x < 3 \quad \text{choose } x \in \{3, 2, 1\} : x < 3
\]

A similar violation of the semantics of TLA\(^+\) exists with \textit{case} expressions, whose
semantics are defined (in Section 15.1.3) in terms of \textit{choose}.

TLC Version 1 the following additional limitations, most of which should be
fixed in version 2. Telling us which ones you find to be a nuisance will help
ensure that they really are fixed in Version 2.

- TLC doesn’t handle specifications that use the \texttt{INSTANCE} statement.
• TLC doesn’t properly handle Cartesian products of more than two sets. Thus, instead of writing $S \times T \times U$, you have to write 
\[
\{ (s, t, u) : s \in S, t \in T, u \in U \}
\]
• You cannot put infix, prefix, or postfix operators in the `CONSTANT` section of the configuration file. Hence, neither of the following replacements may appear in the configuration file:
\[
++ \leftarrow \text{Foo} \quad \text{Bar} \leftarrow \&
\]
• You cannot use an infix, prefix, or postfix operator as the argument of a higher-order operator. For example, if `IsPartialOrder` is the operator described on pages 70–71, you cannot write `IsPartialOrder(<, Nat)`. To get around this problem, you can define 
\[
\text{LessThan}(a, b) \triangleq a < b
\]
and then write `IsPartialOrder(LessThan, Nat)`.
• The Java class that overrides the standard `Bags` module does not implement the `SubBag` operator.
• TLC does not check that a symbol is defined before it is used. This means that TLC will accept illegal recursive operator definitions like 
\[
\text{Silly}(\text{foo}) \triangleq \text{Silly}(\text{foo} + 1)
\]
• The Java classes that override the `Naturals` and `Integers` modules produce incorrect results, rather than an error message, if a computation yields a number outside the interval $-2^{31} \ldots (2^{31} - 1)$.
• TLC doesn’t implement the $\cdot$ (action composition) operator.
• TLC treats strings as primitive values, and not as functions. It thus considers the legal TLA+ expression “abc” [2] to be an error. It also handles only strings containing letters, numbers, spaces, and the following ASCII characters:
\[
< > ? , . / : ; [ ] { } | ~ @ # $ % ^ & * ( ) _ - + =
\]
• TLC allows constants to be replaced by nonconstants operators. Doing so may cause TLC to produce incorrect results. (Version 1 also allows nonconstants to be replaced; this might not be allowed in Version 2.)

13.5 Future Plans

The following additions and improvements to future versions of TLC are being considered. It is unlikely that they will all be implemented; recommendations of which ones are most important would be useful.
• Have the progress reports contain separate counts of generated states that do and don’t satisfy the constraint.

• Introduce subclasses of model values, where a model value is comparable only with model values in its own subclass. This requires also adding some way of handling the construct \( \text{choose } x : x \notin S \).

• Improve debugging information as follows:
  – Print out the context at the point of the error—including variable values, and the values of bound variables.
  – Identify the disjunct of the next-state action responsible for each step when printing a behavior.
  – More precisely identify the place within the specification where an error occurs.
  – When an invariant is found to be false, have TLC print out why it is false—that is, which conjunct or conjuncts are false.

• Add a debugging mode in which TLC won’t stop for an error, but will just keep going.

• Add some coverage analysis, indicating which parts of the next-state relation never evaluated to \( \text{true} \).

• Add some support for using symmetry to reduce the state space searched.

• Have TLC check all assumptions.

13.6 The Fine Print: What TLC Really Does

We now describe in more detail what TLC does. This section is for the intellectually curious and mathematically sophisticated. Most users will not want to read it.

This section specifies the results produced by TLC, not the actual algorithms it uses to compute those results. Moreover, it specifies only what TLC guarantees to do. TLC may actually do better, producing a correct result when the specification allows it to report an error.

13.6.1 TLC Values

Section 13.2.1 (page 122) describes TLC values. That description was incomplete in two ways. First, it did not precisely define when values are comparable. The precise definition is that two TLC values are comparable iff the following rules imply that they are:
1. Two primitive values are comparable if they have the same value type.
   This rule implies that “abc” and “123” are comparable. But “abc” and 123 are not comparable.

2. A model value is comparable with any value. (It is equal only to itself.)

3. Two sets are comparable if they have different numbers of elements, or if they have the same numbers of elements and all the elements in one set are comparable with all the elements in the other.
   This rule implies that \{1\} and \{“a”, “b”\} are comparable and that \{1, 2\} and \{2, 3\} are comparable. However, \{1, 2\} and \{“a”, “b”\} are not comparable.

4. Two functions \(f\) and \(g\) are comparable if (i) their domains are comparable and (ii) if their domains are equal, then \(f[x]\) and \(g[x]\) are comparable for every element \(x\) in their domains.
   This rule implies that \((1, 2)\) and \((“a”, “b”, “c”)\) are comparable, and that \((1, “a”)\) and \((2, “bc”)\) are comparable. However, \((1, 2)\) and \((“a”, “b”)\) are not comparable.

Section 13.2.1 also failed to mention that additional primitive value types can be introduced by Java classes that override modules. The primitive TLC value types are thus Booleans, integers, strings, model values, and any additional value types introduced by overriding.

13.6.2 Overridden Values

As explained in Section 13.2.4 above, an extended module may be overridden by a Java class. All the values and operators defined by the module are replaced by special overridden values. For example, the Java class that overrides the Naturals module replaces \(Nat\) and + by overridden values. An overridden module may not have parameters.

13.6.3 Expression Evaluation

TLC evaluates expressions when performing three different tasks:

- It evaluates the initial predicate to compute the set of initial states.
- It evaluates the next-state action to compute the set of states reachable from a given state by a step of that action.
- It evaluates an invariant when checking that a reachable state is correct.
The result of successfully evaluating an expression is a TLC value. Evaluation may fail, in which case TLC reports an error and halts.

TLC evaluates an expression in a context, which is an assignment of meanings to user-defined symbols, parameters, variables, and primed variables. A meaning is a TLA+ expression formed from the built-in operators of TLA+, TLC values, and overridden values. Variables and primed variables can be assigned only TLC values. For example, consider a module that has two variables \( x \) and \( y \).

To compute the next states starting in a state with \( x = 2 \) and \( y = \{a\} \), TLC evaluates the next-state relation in the context \( C \) in which the defined symbols have the meanings assigned to them by the module, the module’s parameters have the values assigned to them by the configuration file, \( x \) is assigned the value \( 2 \), \( y \) is assigned the value \( \{a\} \), and \( x' \) and \( y' \) have no meanings assigned to them.

We now define the result \( V(e) \) of evaluating a TLA+ expression \( e \), which is a TLC value. If the expression \( e \) does not denote a TLC value, then its evaluation fails. We use an inductive definition that defines \( V(e) \) in terms of two other operations: one-step evaluation \( O(e) \) and set enumeration \( E(e) \) of an expression \( e \). While \( V(e) \) is a TLC value, \( O(e) \) is an arbitrary TLA+ expression and \( E(e) \) is a finite sequence of arbitrary TLA+ expressions. One-step evaluation and enumeration are explained below.

In TLA+, the "\( h \)" in an expression \( e.h \) or in the "\( ! \)" clause of an EXCEPT construct just means \( [\!h!] \). To evaluate an expression, TLC first replaces all such instances of "\( h \)" by \( [\!h!] \).

The inductive rules for computing \( V(e) \) are given below. The rules are stated somewhat informally, using TLA+ notation and the operators from the standard Naturals and Sequences modules (see Chapter 17). Evaluation fails if the rules do not imply that \( V(e) \) is a TLC value. For example, evaluation of the nonsensical expression \( \{1, 2\}[3] \) fails because no rule applies to an expression of this form. Evaluation of \( e \) also fails if a rule implies that computing \( V(e) \) requires evaluating an expression whose evaluation fails.

**Quantification**

The rules for quantifiers use the operator \( \text{Perm} \), where \( \text{Perm}(s) \) is an arbitrary permutation of a sequence \( s \). It can be defined by

\[
\text{Perm}(s) \triangleq \text{LET } Pi \triangleq \text{CHOOSE } f \in [1..\text{Len}(s) \rightarrow 1..\text{Len}(s)] : \forall n \in 1..\text{Len}(s) : \exists m \in 1..\text{Len}(s) : f[m] = n \\
\text{IN } [i \in 1..\text{Len}(s) \rightarrow s[Pi[i]]]
\]

\(5\)This defines \( \text{Perm}(s) \) in terms of a permutation of the elements that depends only on the length of \( s \), not on \( s \) itself. A more precise definition would use the \textit{Choice} operator described on page 188 to make \( \text{Perm}(s) \) depend on \( s \).
For example, \( Perm(\langle 1, 2, 3 \rangle) \) might equal \( (2, 1, 3) \). (Remember that \( E(S) \) is a finite sequence of TLA\(^+\) expressions.)

\[
\forall x \in S : p = \\
\text{LET } s \triangleq Perm(E(S)) \\
AV[n \in Nat] \triangleq \begin{cases} 
\text{true} & \text{if } n = 0 \\
\text{false} & \text{if } \forall (\text{LET } x \triangleq s[n] \text{ in } p) \\
\text{then } AV[n-1] \\
\text{else } \text{false}
\end{cases} \\
\text{in } AV[\text{Len}(s)]
\]

\[
\exists x \in S : p = \\
\text{LET } s \triangleq Perm(E(S)) \\
EV[n \in Nat] \triangleq \begin{cases} 
\text{false} & \text{if } n = 0 \\
\text{true} & \text{if } \exists (\text{LET } x \triangleq s[n] \text{ in } p) \\
\text{then } EV[n-1] \\
\text{else } EV[n-1]
\end{cases} \\
\text{in } EV[\text{Len}(s)]
\]

**One-Step Evaluable Expressions**

The result \( O(e) \) of one-step evaluation is a TLA\(^+\) expression that is obtained by performing just the first step in the evaluation of \( e \). For example, suppose \( e \) is the expression

\[
\text{if } x > 2 \text{ then } x + 2 \text{ else } x - 2
\]

In a context in which \( x \) is assigned the value 3, \( O(e) \) is the expression \( x + 2 \). This differs from the result \( V(e) \) of evaluating \( e \), which is the TLC value 5. TLC may be able to perform one-step evaluation even if it can’t evaluate the expression. For example, one-step evaluation of

\[
\text{if } x > 2 \text{ then } \{ j \in Nat : j > i \} \text{ else } \{
\}
\]

in a context in which \( x \) equals 3 is the expression \( \{ j \in Nat : j > 5 \} \). TLC will fail if it tries to evaluate this expression because it denotes infinite set, which is not a TLC value.

An expression \( e \) is **one-step evaluable** if \( O(e) \) is defined. The following kinds of expressions are one-step evaluable: \( \text{LET, IF/THEN/ELSE, CASE, CHOOSE, and function application (expressions of the form } f[e] \text{)} \). The rule for evaluating any one-step evaluable expression \( e \) is \( V(e) = V(O(e)) \). In other words, a one-step evaluable expression is evaluated by evaluating the result of one-step evaluation.
Below are the rules for one-step evaluation for all of these except function application, whose rules are given later. (The operator \( Perm \), appearing in the rule for \( \text{choose} \), is defined above.)

\[
\mathcal{O}(\text{let } x \overset{\Delta}{=} v \text{ in } e) \quad \text{evaluated in the current context is equal to } e, \text{ with the evaluation proceeding in the context augmented by assigning } v \text{ to } x.
\]

\[
\mathcal{O}(\text{if } p \text{ then } e_1 \text{ else } e_2) =
\begin{align*}
\text{case } \mathcal{V}(p) &= \text{true} \rightarrow e_1 \\
\Box \mathcal{V}(p) &= \text{false} \rightarrow e_2 \\
\Box \text{other} \rightarrow \text{evaluation fails}
\end{align*}
\]

\[
\mathcal{V}(\text{case } p_1 \rightarrow e_1 \Box \cdots \Box p_n \rightarrow e_n) = \text{let } j \overset{\Delta}{=} \mathcal{V}(\text{choose } i \in 1 \ldots n : p_i) \text{ in } \mathcal{V}(e_i)
\]

\[
\mathcal{V}(\text{case } p_1 \rightarrow e_1 \Box \cdots \Box p_n \rightarrow e_n \Box \text{other} \rightarrow e) =
\begin{align*}
\text{if } \mathcal{V}(\exists i \in 1 \ldots n : p_i) \text{ then } &\mathcal{V}(\text{case } p_1 \rightarrow e_1 \Box \cdots \Box p_n \rightarrow e_n) \\
\text{else } &\mathcal{V}(e)
\end{align*}
\]

\[
\mathcal{O}(\text{choose } x \in S : p) =
\begin{align*}
&\text{let } s \overset{\Delta}{=} \text{Perm}(\mathcal{E}(S)) \\
&CV[n \in \text{Nat}] \overset{\Delta}{=} \text{if } n = 0 \text{ then } \text{evaluation fails} \\
&\text{else if } \mathcal{V}(\text{let } x \overset{\Delta}{=} s[n] \text{ in } p) \\
&\text{then } s[n] \\
&\text{else } CV[n - 1]
\end{align*}
\]

\[
\text{in } CV[\text{Len}(s)]
\]

### Enumerating Set-Valued Expressions

If \( e \) is an expression whose value is a set, then the result \( \mathcal{E}(e) \) of enumerating \( e \) is a finite sequence of TLA+ expressions obtained by enumerating the elements of \( e \) without evaluating them. The order of elements in the sequence doesn’t matter.\(^6\)

For example, if \( e \) is the expression \( \{1 + 1, 2, 3 - 1\} \), then \( \mathcal{E}(e) \) is the sequence \( (1 + 1, 2, 3 - 1) \) of expressions. Note that the sets \( \{1, "a"\} \) and \( \{\text{Nat}, \text{Int}\} \) can be enumerated even though they cannot be evaluated. (The set \( \{1, "a"\} \) isn’t a TLC value because its elements are not comparable; the set \( \{\text{Nat}, \text{Int}\} \) isn’t a TLC value because its elements are not TLC values.)

A set-valued expression is evaluated by enumerating it, then evaluating each expression in the enumeration, and finally eliminating duplicate values. More

\(^6\)It would perhaps be better to define an enumeration to be a bag, but I won’t bother introducing notation for bags here.
precisely, for any set-valued expression $S$:

\[
\forall(S) = \\
\text{LET } n \triangleq \text{Len}(E(S)) \\
s[i \in 0 \ldots n] \triangleq \text{IF } i = 0 \\
\quad \text{THEN } \{\} \\
\quad \text{ELSE IF } \exists j \in 1 \ldots \text{Len}(s[i-1]) : s[i-1][j] = E(S)[i] \\
\quad \quad \text{THEN Append}(s[i-1], E(S)[i]) \\
\quad \quad \text{ELSE } s[i-1]
\]

Below are the rules for enumerating set-valued expressions. They use the `SelectSeq` operator, which is defined in the `Sequences` module so that `SelectSeq(s, Test)` is the subsequence of the sequence $s$ consisting of all elements $e$ such that $Test(e)$ is true.

\[
E(S \cup T) = E(S) \circ E(T) \\
E(S \cap T) = \text{LET } InT(s) \triangleq \forall(s \in T) \\
\quad \text{IN } \text{SelectSeq}(E(S), InT) \\
E(S \setminus T) = \text{LET } NotInT(s) \triangleq \forall(s \notin T) \\
\quad \text{IN } \text{SelectSeq}(E(S), NotInT) \\
E(\{e_1, \ldots, e_n\}) = \langle e_1, \ldots, e_n \rangle \\
E(\{x \in S : p\}) = \text{LET } PTrue(s) \triangleq \forall(\text{LET } x \triangleq s \text{ IN } p) \\
\quad \text{IN } \text{SelectSeq}(E(S), PTrue) \\
E(\{e : x \in S\}) = \{i \in 1 \ldots \text{Len}(E(S)) \mapsto \text{LET } x \triangleq E(S)[i] \text{ IN } e\}
\]

$E(\text{subset } S)$ is a sequence of length $2^{\text{Len}(E(S))}$ whose elements consist of all expressions of the form $\{E(S)[i_1], \ldots, E(S)[i_k]\}$ for all subsets $\{i_1, \ldots, i_k\}$ of $1 \ldots \text{Len}(E(S))$.

$E(\text{union } S) = E(E(S)[1]) \circ \cdots \circ E(E(S)[\text{Len}(E(S))])$

$E(\text{[S -> T]})$ is a sequence of length $\text{Len}(E(T))^{\text{Cardinality}(S)}$ whose elements consist of all expressions of the form

\[
(s_1 : E(T)[i_1] @ @ \cdots @ @ s_n : E(T)[i_n])
\]

for all possible choices of $i_j$ in $1 \ldots \text{Len}(E(T))$, where $V(S) = \{s_1, \ldots, s_n\}$. The enumeration fails if $S$ cannot be evaluated.

$E([h_1 : S_1, \ldots, h_n : S_n])$ is a sequence of length $\text{Len}(E(S_1)) \cdots \text{Len}(E(S_n))$ whose elements consist of all expressions of the form

\[
[h_1 \mapsto E(S_1)[i_1], \ldots, h_1 \mapsto E(S_n)[i_n]]
\]
for all possible choices of $i_j$ in $1 \ldots \text{Len}(\mathcal{E}(S_j))$.

$\mathcal{E}(S_1 \times \cdots \times S_n)$ is a sequence of length $\text{Len}(\mathcal{E}(S_1)) \ast \cdots \ast \text{Len}(\mathcal{E}(S_n))$ whose elements consist of all expressions of the form

$\langle \mathcal{E}(S_1)[i_1], \ldots, \mathcal{E}(S_n)[i_n] \rangle$

for all possible choices of $i_j$ in $1 \ldots \text{Len}(\mathcal{E}(S_j))$.

$\mathcal{E}(\text{domain } f)$ depends as follows on the form of the expression $f$:

$$
\begin{align*}
\mathcal{E}(\text{domain } [x \in S \mapsto e]) & = \mathcal{E}(S) \\
\mathcal{E}(\text{domain } [g \ \text{EXCEPT} \ \ldots]) & = \mathcal{E}(\text{domain } g) \\
\mathcal{E}(\text{domain } (s_1 :> t_1 @ @ \cdots @ @ s_n :> t_n)) & = \{s_1, \ldots, s_n\} \\
\mathcal{E}(\text{domain } (s_1, \ldots, s_n)) & = \{1, \ldots, n\} \\
\mathcal{E}(\text{domain } [h_1 \mapsto e_1, \ldots, h_n \mapsto e_n]) & = \{"h_1", \ldots, "h_n"\}.
\end{align*}
$$

In addition, if $f$ is a one-step evaluable expression, then $\mathcal{E}(\text{domain } f) = \mathcal{E}(\text{domain } O(f))$.

In addition, if $e$ is a one-step evaluable expression, then $\mathcal{E}(e) = \mathcal{E}(O(e))$.

**Boolean-Valued Expressions**

Evaluation of Boolean expressions is defined by the following rules.

$$
\begin{align*}
\mathcal{V}(p \lor q) & = \text{IF } \mathcal{V}(p) \text{ THEN TRUE ELSE IF } \mathcal{V}(q) \text{ THEN TRUE ELSE FALSE} \\
\mathcal{V}(p \land q) & = \text{IF } \mathcal{V}(p) \text{ THEN IF } \mathcal{V}(q) \text{ THEN TRUE ELSE FALSE ELSE FALSE} \\
\mathcal{V}(\lnot p) & = \text{IF } \mathcal{V}(p) \text{ THEN FALSE ELSE TRUE} \\
\mathcal{V}(p \Rightarrow q) & = \text{IF } \mathcal{V}(p) \text{ THEN IF } \mathcal{V}(q) \text{ THEN TRUE ELSE FALSE ELSE TRUE} \\
\mathcal{V}(p \equiv q) & = \text{IF } \mathcal{V}(p) \text{ THEN IF } \mathcal{V}(q) \text{ THEN TRUE ELSE FALSE ELSE IF } \mathcal{V}(q) \text{ THEN FALSE ELSE TRUE} \\
\end{align*}
$$

**Equality**

An expression of the form $v = w$ is evaluated by first evaluating $v$ and $w$. Evaluation of $v = w$ fails if the evaluation of $v$ or $w$ fails, or if the resulting values are not comparable, according to the rules of Section 13.6.1 above. If they are comparable, then equality of TLC values is computed in the obvious way from the equality relation on primitive values—namely, by using the following rules. In these rules, the $s_i$ are assumed to be all distinct, as are the $t_j$. 


\[ \forall \{s_1, \ldots, s_n\} = \{t_1, \ldots, t_m\} = \\
\text{if } n = m \text{ then } \forall (\forall i \in 1 \ldots n : \exists j \in 1 \ldots n : s_i = t_j) \\
\text{else false} \]

\[ \forall((s_1:> d_1 @@ \cdots @@ s_n:> d_n) = (t_1:> e_1 @@ \cdots @@ t_m:> e_m)) = \\
\text{if } \forall(\{s_1, \ldots, s_n\} = \{t_1, \ldots, t_m\}) \\
\text{then } \forall i \in 1 \ldots n : \text{let } j =_V \forall(\text{choose } k \in 1 \ldots m : s_i = t_k) \\
\text{in } \forall(e_i = d_j) \\
\text{else false} \]

**Set Membership**

The value of an expression of the form \( v \in w \) depends as follows on the form of \( w \).

\[ \forall(v \in S \cup T) = \text{if } \forall(v \in S) \text{ then true else } \forall(v \in T) \]
\[ \forall(v \in S \cap T) = \text{if } \forall(v \in S) \text{ then } \forall(v \in T) \text{ else false} \]
\[ \forall(v \in S \setminus T) = \text{if } \forall(v \in S) \text{ then } \forall(v \in T) \text{ then false else true} \\
\text{else false} \]
\[ \forall(v \in \{e_1, \ldots, e_n\}) = \forall(\exists s \in \{e_1, \ldots, e_n\} : v = s) \]
\[ \forall(v \in \{x \in S : p\}) = \text{if } \forall(v \in S) \text{ then } \forall(\text{let } x =_V v \text{ in } p) \\
\text{else false} \]
\[ \forall(v \in \{e : x \in S\}) = \forall(\exists s \in S : v = \text{let } x =_V s \text{ in } e) \]
\[ \forall(v \in \text{subset } S) = \forall(\forall s \in v : s \in S) \]
\[ \forall(v \in \text{union } S) = \forall(\forall s \in S : v \in s) \]
\[ \forall(v \in \text{domain } f) = \forall(v \in \forall(v \in \text{domain } f)), \text{ except in the following two cases:} \]

\[ \forall(v \in \text{domain } [x \in S \mapsto e]) = \forall(v \in S) \]
\[ \forall(v \in \text{domain } [g \text{ except } \ldots]) = \forall(v \in \text{domain } g) \]
\[ \forall(v \in [S \rightarrow T]) = \text{if } \forall(\text{domain } v = S) \text{ then } \forall(\forall s \in S : v[s] \in T) \\
\text{else false} \]
\[ \forall(v \in [h_1 : S_1, \ldots, h_n : S_n]) = \text{if } \forall(\text{domain } v = \{h_1, \ldots, h_n\}) \\
\text{then } \forall(i \in 1 \ldots n : v[h_i] \in S_i) \]
\[ \text{else false} \]
\[ \forall(v \in S_1 \times \ldots \times S_n) = \text{if } \forall(\text{domain } v = 1 \ldots n) \\
\text{then } \forall(i \in 1 \ldots n : v[i] \in S_i) \]
\[ \text{else false} \]
If \( w \) is one of a class of special set constants, then TLC evaluates \( v \in w \) by determining if \( \mathcal{V}(v) \) is an element of the set represented by \( w \). For example, \( \mathcal{V}(v \in \text{STRING}) \) equals TRUE if \( \mathcal{V}(v) \) is a string and it equals FALSE if \( \mathcal{V}(v) \) is a model value; otherwise, evaluation fails. The built-in special set constants are STRING, BOOLEAN. An overridden symbol can also be a special set constant. In particular, the standard Java classes that override the Naturals and Integers modules define Nat and Int to be special set constants.

### Function Application

As stated above, an expression of the form \( v[w] \) is one-step evaluable. Here are the rules that define \( \mathcal{O}(v[w]) \) for the possible function-valued expressions \( w \).

\[
\mathcal{O}(\{x \in S \mapsto e\}[w]) = \begin{cases} \text{IF } w \in S \text{ THEN LET } x \triangleq w \text{ IN } e & \text{ELSE evaluation fails} \\ \end{cases}
\]

\[
\mathcal{O}(\{f \text{ EXCEPT } ![e_1] \ldots ![e_n] = d\}[w]) = \begin{cases} \text{IF } w \in \text{DOMAIN } f & \text{THEN} \\ \text{IF } n = 1 & \text{THEN } d \\ \text{ELSE } [f \text{ EXCEPT } ![e_2] \ldots ![e_n] = d] & \text{ELSE evaluation fails} \\ \end{cases}
\]

\[
\mathcal{O}(\{h_1 \mapsto e_1, \ldots, h_n \mapsto e_n\}[w]) = \begin{cases} \text{LET } j \triangleq \text{CHOOSE } i \in 1 \ldots n : w = h_i & \text{IN } e_j \\ \end{cases}
\]

\[
\mathcal{O}(\langle e_1, \ldots, e_n \rangle[w]) = \begin{cases} \text{IF } u \in 1 \ldots n & \text{THEN } e_u \\ \text{ELSE evaluation fails} , \\ \end{cases}
\]

where \( u \) is the value obtained by evaluating \( w \).

\[
\mathcal{O}(\langle s_1 : \mapsto e_1 \@ \@ \ldots \@ \@ s_n : \mapsto e_n \rangle[w]) = \begin{cases} \text{IF } \mathcal{V}(\exists i \in 1 \ldots n : w = s_i) & \text{THEN LET } j \triangleq \mathcal{V}(\text{CHOOSE } i \in 1 \ldots n : w = s_i) \\ \text{IN } e_j \\ \text{ELSE evaluation fails} \end{cases}
\]

If \( v \) is one-step evaluable, then \( \mathcal{O}(v[w]) = \mathcal{O}(\mathcal{O}(v)[w]) \).

### Functions and Records

\[
\mathcal{V}(\{x \in S \mapsto e\}) = (s_1 : \mapsto (\text{LET } x \triangleq s_1 \text{ IN } e) \@ \@ \cdots \@ \@ s_n : \mapsto (\text{LET } x \triangleq s_n \text{ IN } e)),
\]

where \( \mathcal{V}(S) \) equals \( \{s_1, \ldots, s_n\} \).
\( \mathcal{V}([f \; \text{except} \; ![e_1] \ldots ![e_n] = d]) =
\) 
\( (\ldots @@ s_i := (\text{if } \mathcal{V}(s_i = e_i)
\text{then if } n = 1
\text{then } \mathcal{V}(\text{let } \overset{=}{} = t_i \text{ in } d)
\text{else } \mathcal{V}([t_i \; \text{except} \; ![e_2] \ldots ![e_n] = d])
\text{else } t_i
\ldots ),
\) 
where \( \mathcal{V}(f) \) equals \( (\ldots @@ s_i := t_i @@ \ldots ). \)
\( \mathcal{V}([h_1 \mapsto e_1, \ldots, h_n \mapsto e_n]) = \) \( (\text{“}h_1' \mapsto \mathcal{V}(e_1) @@ \ldots @@ \text{“}h_n' \mapsto \mathcal{V}(e_n)) \)
\( \mathcal{V}((e_1, \ldots, e_n)) = \) \( (1 \mapsto \mathcal{V}(e_1) @@ \ldots @@ n \mapsto \mathcal{V}(e_n)) \)

**Other Constant Operators**
\( \mathcal{V}(v \neq w) = \mathcal{V}(\neg(v = w)) \)
\( \mathcal{V}(e \notin S) = \mathcal{V}(\neg(e \in S)) \)
\( \mathcal{V}(S \subseteq T) = \mathcal{V}(\forall s \in S : s \in T) \)

**Primitive Values and Overridden Operators**
If \( v \) is a primitive or overridden value, then \( \mathcal{V}(v) = v. \)
If \( Op \) is an overridden operator, then \( \mathcal{V}(Op(e_1, \ldots, e_n)) \) equals
\( Op(\mathcal{V}(e_1), \ldots, \mathcal{V}(e_n)) \)
if this is a TLC value. If it isn’t, then evaluation fails. For example, if the
concatenation operator \( \circ \) of the \textit{Sequences} module is overridden, then \( \mathcal{V}(s \circ t) \)
equals \( \mathcal{V}(s) \circ \mathcal{V}(t), \) which is the function
\( (1 := \mathcal{V}(s[1]) @@ \ldots @@ \mathcal{V}(\text{Len}(s)) := \mathcal{V}(s[\text{Len}(s)]) @@
(1 + \mathcal{V}(\text{Len}(s))) := \mathcal{V}(t[1]) @@ \ldots @@
\mathcal{V}(\text{Len}(s)) + \mathcal{V}(\text{Len}(t)) := \mathcal{V}(t[\text{Len}(t)]) ) \)
if \( \mathcal{V}(s) \) and \( \mathcal{V}(t) \) are sequences.

**Action Expressions**
\( \mathcal{V}(e') \) in a context \( C \) equals \( \mathcal{V}(e) \) in the context obtained from \( C \) by assigning
to each unprimed variable \( x \) the value assigned by \( C \) to \( x' \), and assigning no
value to any primed variable.
\( \mathcal{V}([A]_c) = \mathcal{V}(A \lor e' = e) \)
\( \mathcal{V}((A)_c) = \mathcal{V}(A \land e' \neq e) \)
**13.6. THE FINE PRINT: WHAT TLC REALLY DOES**

\( V(\text{Enabled } A) \) in a context \( C \) is computed by attempting to find a state \( t \) such that \( s \to t \) is an \( A \) step, where \( s \) is the state that assigns to each variable the value assigned to it by \( C \). The value of \( V(\text{Enabled } A) \) is \textit{true} if such a \( t \) exists, otherwise it is \textit{false}.

\( V(A \cdot B) \) in a context \( C \) is computed by finding all states \( t \) such that \( s \to t \) is an \( A \) step, where \( s \) is the state that assigns to each variable the value assigned to it by \( C \), then finding all states \( u \) such that \( t \to u \) is a \( B \) step. Note that TLC evaluates an action only when either computing the next-state action to find successor states to a state \( s \), or when evaluating an \textit{enabled} expression.

### 13.6.4 Fingerprinting

The description of TLC’s computation in Section 13.2.6 is inaccurate in one important respect. TLC does not actually keep the set of states \( \mathcal{R} \). Instead, it keeps the set of fingerprints of those states. A fingerprint of a state is a 64-bit value generated by a “hashing” function. Ideally, the probability that two different states have the same fingerprint is \( 2^{-64} \), which is a very small number.

Rather than checking if a state \( s \) is in \( \mathcal{R} \), TLC checks if its fingerprint equals the fingerprint of a state already in \( \mathcal{R} \). With very high probability, that will be true iff \( s \) is in \( \mathcal{R} \). However, it is possible for a \textit{collision} to occur, meaning that the fingerprint of \( s \) equals the fingerprint of some other state in \( \mathcal{R} \). If this happens, TLC will not explore the successor states of \( s \), and thus may not find all reachable states.

It is essentially impossible to do an accurate \textit{a posteriori} calculation of the probability that TLC did not check all reachable states because of a fingerprint collision. If the probability that any two states have the same fingerprint is \( 2^{-64} \), then a simple calculation shows that if TLC generated \( n \) states with \( m \) distinct fingerprints, then the probability of a collision is about \( m \cdot (n - m) \cdot 2^{-64} \). However, the process of generating states is highly nonrandom, and no known fingerprinting scheme can guarantee that the probability of any two distinct states generated by TLC having the same fingerprint is actually \( 2^{-64} \). So, this estimate is an optimistic one.

Another way to estimate the probability of collision is empirical. If there was a collision, then it is likely that there was also a “near miss”. One estimate of the probability that there was a collision is the maximum value of \( 1/|f_1 - f_2| \) over all pairs \( \langle f_1, f_2 \rangle \) of distinct fingerprints generated by TLC for the states that it found.

TLC prints out both probability estimates when it finishes. You can use these values to decide how much confidence to place in the completeness of TLC’s exploration of the reachable states. You may wish to supplement TLC’s model checking calculation with random simulation, which does not maintain the set \( \mathcal{R} \) and thus uses no fingerprinting.
Part IV

The TLA$^+$ Language
This part describes version 1.0 of TLA+. Actually, it describes only the subset of TLA+ used for writing specifications. TLA+ contains additional constructs, used only for writing proofs and proof rules, that are not described here.

Chapter 14 describes the syntax of TLA+. It is a reference manual for the grammar. Chapters 15 and 16 describe the semantics of TLA+. They are heavy going and are for sophisticated readers who want to understand the fine points of the language’s semantics.
Chapter 14

The Syntax of TLA$^+$

The term *syntax* has two different usages, which I will somewhat arbitrarily attribute to mathematicians and computer scientists. A computer scientist would say that \( \langle a, a \rangle \) is a syntactically correct TLA$^+$ expression. A mathematician would say that the expression is syntactically correct iff it appears in a context in which \( a \) is defined or declared. A computer scientist would call this requirement a *semantic* rather than a syntactic condition. A mathematician would say that \( \langle a, a \rangle \) is meaningless if \( a \) isn’t defined or declared, and one can’t talk about the semantics of a meaningless expression.

This chapter describes the syntax of TLA$^+$, in the computer scientist’s sense of syntax. (The “semantic” part of the syntax is specified in Chapter 16.) TLA$^+$ is designed to be easy for humans to read and write. In particular, its syntax for expressions tries to capture some of the richness of ordinary mathematical notation. This makes a precise specification of the syntax rather complicated. Such a specification has been written in TLA$^+$, but it is quite detailed and you probably don’t want to look at it unless you are writing a parser for the language. This chapter gives a less formal description of the syntax that should answer any questions likely to arise in practice. Section 14.1 specifies precisely a simple grammar that ignores some aspects of the syntax such as operator precedence, indentation rules for \( \wedge \) and \( \vee \) lists, and comments. These other aspects are explained informally in Section 14.2.1. Finally, Section 14.3 lists the correspondence between the typed and typeset versions of symbols, as well as all user-definable operator symbols.
14.1 The Simple Grammar

The simple grammar of TLA\(^+\) is described in BNF. Just for fun, this BNF grammar is specified in TLA\(^+\). Specifying a BNF grammar in TLA\(^+\) provides a nice little exercise in “mathemticizing” a simple concept. Moreover, it demonstrates the flexibility of ordinary mathematics, as formalized in TLA\(^+\). The syntax of TLA\(^+\) doesn’t permit us to write BNF grammars exactly the way you’re used to seeing them, but we can come reasonably close.

Section 14.1.1 explains how to specify a BNF grammar. The BNF grammar for TLA\(^+\) is specified in Section 14.1.2 as a TLA\(^+\) module, with comments that describe how to read the TLA\(^+\) specification as an ordinary BNF grammar. If you are familiar with BNF grammars and just want to learn the syntax of TLA\(^+\), you can skip directly to Section 14.1.2 (page 168).

14.1.1 BNF Grammars

Let’s start by reviewing BNF grammars. Consider the little language SE of simple expressions described by this BNF grammar:

\[
\begin{align*}
expr & ::= \text{ident} \mid expr \ op \ expr \mid (expr) \mid \text{let def in expr} \\
def & ::= \text{ident} \ == \ expr
\end{align*}
\]

where \(op\) is some class of infix operators like +, and \text{ident} is some class of identifiers such as \(abc\) and \(x\). The language SE contains expressions like

\[abc \ + \ (\text{let } x \ == \ y + abc \ \text{in } x \ * \ x)\]

I will represent this expression as the sequence

\[
\langle \text{“abc”}, \text{“+”}, \text{“(”}, \text{“LET”}, \text{“x”}, \text{“==”}, \text{“y”}, \text{“+”}, \text{“abc”}, \text{“IN”}, \text{“x”}, \text{“*”}, \text{“x”}, \text{“)”} \rangle
\]

of strings. The strings such as “abc” and “+” appearing in this sequence are usually called lexemes. In general, a sequence of lexemes is called a sentence; and a set of sentences is called a language. So, we want to define the language SE to consist of the set of all such sentences described by the BNF grammar.\footnote{BNF grammars are also used to specify how an expression is parsed—for example that \(a + b * c\) is parsed as \((a + (b * c))\) rather than \((a + b) * c\). By considering the grammar to specify only a set of sentences, we are deliberately not capturing that use in our TLA\(^+\) representation of BNF grammars.}

To represent a BNF grammar in TLA\(^+\), we must assign a mathematical meaning to nonterminal symbols like def, to terminal symbols like op, and to the grammar’s two productions. The method that I find simplest is to let the meaning of a nonterminal symbol be the language that it generates. Thus, the meaning of expr is the language SE itself. I define a grammar to be a function \(G\) such that, for any string \(\text{“str”}\), the value of \(G[\text{“str”}]\) is the language generated by
the nonterminal \textit{str}. Thus, if \( G \) is the BNF grammar above, then \( G[\text{"expr"}] \) is the complete language \( \text{SE} \), and \( G[\text{"def"}] \) is the language defined by the production for \textit{def}, which contains sentences like

\[
\{ \text{"y"}, \text{"=="}, \text{"qq"}, \text{"+"}, \text{"abc"} \}
\]

Instead of letting the domain of \( G \) consist of just the two strings \text{"expr"} and \text{"def"}, it turns out to be more convenient to let its domain be the entire set \text{STRING} of strings, and to let \( G[s] \) be the empty language (the empty set) for all strings \( s \) other than \text{"expr"} and \text{"def"}.

When discussing the mathematical meaning of records, Section 5.2 explains that \textit{r.ack} is an abbreviation for \( r[\text{"ack"}] \). This is the case even if \( r \) isn’t a record. So, we can write \( G\.op \) instead of \( G[\text{"op"}] \). (A grammar isn’t a record because its domain is the set of all strings rather than a finite set of strings.)

A terminal like \textit{ident} can appear anywhere to the right of a \textit{::=} that a nonterminal like \textit{expr} can, so a terminal should also be a set of sentences. A terminal is a set of sentences each containing a single lexeme. I will call such a sentence a \textit{token}. Thus, the terminal \textit{ident} is a set containing tokens such as \( \langle \text{"abc"} \rangle \), \( \langle \text{"x"} \rangle \), and \( \langle \text{"qq"} \rangle \). Any terminal appearing in the BNF grammar must be represented by a set of tokens, so the \textit{==} in the grammar for \text{SE} is the set \( \{ \langle \text{"=="} \rangle \} \). Let’s define the operator \textit{tok} by

\[
\text{tok}(s) \triangleq \{ \langle s \rangle \}
\]

so we can write this set of tokens as \( \text{tok}(\text{"=="}) \).

A production expresses a relation between the values of \textit{G.str} for some grammar \( G \) and some strings \textit{str}. For example, the production

\[
def ::= \text{ident} \text{== expr}
\]

asserts that a sentence \( s \) is in \( G.def \) iff it has the form \( i \circ \langle \text{"=="} \rangle \circ e \) for some token \( i \) in \textit{ident} and some sentence \( e \) in \( G.expr \). In mathematics, a formula about \( G \) must mention \( G \) (perhaps indirectly by using a symbol defined in terms of \( G \)). So, we can try writing this production in TLA+ as

\[
G.def ::= \text{ident} \text{tok(\"==\")} \text{G.expr}
\]

In the expression to the right of the ::=, adjacency is expressing some operation. Just as we have to make multiplication explicit by writing \( 2 \times x \) instead of \( 2x \), we must express this operation by an explicit operator. Let’s use \& to write this production as

\[
G.def ::= \text{ident} \& \text{tok(\"==\")} \& G.expr
\]

This expresses the desired relation between the sets \( G.def \) and \( G.expr \) of sentences if ::= is defined to be equality and \& is defined so that \( L \& M \) is the
set of all sentences obtained by concatenating a sentence in \( L \) with a sentence in \( M \):
\[
L \sqcup M \triangleq \{ s \circ t : s \in L, t \in M \}
\]
The production
\[
expr ::= \text{ident} \mid expr \ op \ expr \mid ( expr ) \mid \text{LET def IN expr}
\]
can similarly be expressed as
\[
G.\ expr ::= \text{ident} \mid \ G.\ expr \ & \ op \ & \ G.\ expr \mid \text{tok(""} \ & \ G.\ expr \ & \ \text{tok(""}) \mid \text{tok("LET")} \ & \ G.\ def \ & \ \text{tok("IN")} \ & \ G.\ expr
\]
This expresses the desired relation if \( \circ \) (which means \textbf{or} in the BNF grammar) is defined to be set union \((\cup)\).

We can also define the following operators that are sometimes used in BNF grammars:

- \textbf{Nil} is defined so that \( \text{Nil} \ & S \) equals \( S \) for any set \( S \) of sentences:
  \[
  \text{Nil} \triangleq \{()\}
  \]
- \( L^+ \) equals \( L \mid L \ & L \mid L \ & L \ & L \mid \ldots:\)
  \[
  L^+ \triangleq \text{LET } LL[n \in \text{Nat}] \triangleq \begin{array}{l}
  \text{IF } n = 0 \text{ THEN } L \\
  \text{ELSE } LL[n - 1] \mid LL[n - 1] \ & L \\
\end{array}
  \text{IN UNION } \{LL[n] : n \in \text{Nat}\}
  \]
- \( L^* \) equals \( \text{Nil} \mid L \mid L \ & L \mid L \ & L \ & L \mid \ldots: \)
  \[
  L^* \triangleq \text{Nil}|L^+
  \]
The BNF grammar for \( SE \) consists of two productions. Each production is expressed in \( TLA^+ \) as a formula. The entire grammar is the single formula that is the conjunction of those two formulas. We must turn this formula into a mathematical definition of a grammar \( GSE \), which is a function from strings to languages. The formula is an assertion about a grammar \( G \). We define \( GSE \) to be the smallest grammar \( G \) satisfying this formula, where grammar \( G_1 \) smaller than \( G_2 \) means that \( G_1[s] \subseteq G_2[s] \) for every string \( G \). We define an operator \textbf{LeastGrammar} so that \( \text{LeastGrammar}(P) \) is the smallest grammar \( G \) satisfying \( P(G) \):

\[
\text{LeastGrammar}(P(\_)) \triangleq \text{CHOOSE } G \in \text{Grammar : } \land P(G) \land \forall H \in \text{Grammar : } P(H) \Rightarrow (\forall s \in \text{STRING : } G[s] \subseteq H[s])
\]
Letting \( P(G) \) be the conjunction of the two formulas above, we can define the grammar \( GSE \) to be \( \text{LeastGrammar}(P) \). We can then define the language \( SE \) to equal \( \text{GSE} \text{.expr} \). The smallest grammar \( G \) satisfying a formula \( P \) must have \( G[s] \) equal to the empty language for any string \( s \) that doesn’t appear in \( P \). Thus, \( GSE[s] \) equals the empty language \( \{ \} \) for any string \( s \) other than “expr” and “def”.

To complete our specification of \( GSE \), we must define the sets \( \text{ident} \) and \( \text{op} \) of tokens. We can define the set \( \text{op} \) of operators by enumerating them—for example:

\[
\text{op} \triangleq \text{tok}(\text{"+"}) \mid \text{tok}(\text{"-"}) \mid \text{tok}(\text{"*"}) \mid \text{tok}(\text{"/"})
\]

To express this a little more compactly, let’s define \( \text{Tok}(S) \) to be the set of all tokens formed from elements in the set \( S \) of lexemes:

\[
\text{Tok}(S) \triangleq \{ (s) : s \in S \}
\]

We can then write

\[
\text{op} \triangleq \text{Tok}\{\text{"+"}, \text{"-"}, \text{"*"}, \text{"/"}\}
\]

Let’s define \( \text{ident} \) to be the set of tokens whose lexemes are words made entirely of lower-case letters, such as “abc”, “qq”, and “x”. To learn how to do that, we must first understand what strings in TLA+ really are. In TLA+, a string is a sequence of characters. (We don’t care, and the semantics of TLA+ don’t specify, what a character is.) We can therefore apply the usual string operators on them. For example, Tail(“abc”) equals “bc”, and “abc” o “de” equals “abcde”.

The operators like \( \& \) that we just defined for expressing BNF were applied to sets of sentences, where a sentence is a sequence of lexemes. These operators can be applied just as well to sets of sequences of any kind—including sets of strings. For example, \{“one”, “two”\} \& \{“s”\} equals \{“ones”, “twos”\}, and \{“ab”\}^+ is the set consisting of all the strings “ab”, “abab”, “ababab”, etc. So, we can define \( \text{ident} \) to equal \( \text{Tok}(\text{Letter}^+) \), where \( \text{Letter} \) is the set of all lexemes consisting of a single lower-case letter:

\[
\text{Letter} \triangleq \{ “a”, “b”, \ldots , “z” \}
\]

Writing this definition out in full (without the “…” ) is tedious. We can make this a little easier as follows. We first define the operator \( \text{OneOf}(s) \) to be the set of all one-character strings made from the characters of the string \( s \):

\[
\text{OneOf}(s) \triangleq \{ [i \in \{1\} \mapsto s[i]] : i \in \text{DOMAIN } s \}
\]

We can then define

\[
\text{Letter} \triangleq \text{OneOf}(\text{“abcdefghijklmnopqrstuvwxyz”})
\]
The complete definition of the grammar GSE appears in Figure 14.1 on this page.

All the operators we’ve defined here for specifying grammars are grouped into module BNFGrammars, which appears in Figure 14.2 on the next page.

14.1.2 The BNF Grammar of TLA^+

The following TLA^+ module specifies the simple BNF grammar of TLA^+. It makes use of the operators explained in Section 14.1.1. above, which are in the BNFGrammars module. However, if you are already familiar with BNF grammars, you should be able to read it without having read these previous sections.

```plaintext
MODULE TLAPlusGrammar
extends Naturals, Sequences, BNFGrammars

This module defines a simple grammar for TLA^+ that ignores many aspects of the language such as operator precedence and indentation rules. I use the term sentence to mean a sequence of lexemes, where a lexeme is just a string. The BNFGrammars module defines the following standard conventions for writing sets of sentences: L | M means an L or an M, L* means the concatenation of zero or more Ls, and L+ means the concatenation of one or more Ls. The concatenation of an L and an M is denoted by L & M rather than the customary juxtaposition LM. Nil is the null sentence, so Nil & L equals L for any L.

A token is a one-lexeme sentence. There are two operators for defining sets of tokens: if s is a lexeme, then tok(s) is the set containing the single token ⟨s⟩; and if S is a set of lexemes, then Tok(S) is the set containing all tokens ⟨s⟩ for s ∈ S. In comments, I will identify the token ⟨s⟩ with the string s.

We begin by defining two useful operators. First, a CommaList(L) is defined to be an L or a sequence of L’s separated by commas.

CommaList(L) ≡ L & tok(“,”) & L*

Next, if c is a character, then we define AtLeast4(“c”) to be the set of tokens consisting of 4 or more c’s.
```

Figure 14.1: The definition of the grammar GSE for the language SE.
14.1. THE SIMPLE GRAMMAR

MODULE BNFGrammars

A sentence is a sequence of strings. (In standard terminology, the term “lexeme” is used instead of “string”.) A token is a sentence of length one—that is, a one-element sequence whose single element is a string. A language is a set of sentences.

LOCAL INSTANCE Naturals, Sequences

OPERATORS FOR DEFINING SETS OF TOKENS

OneOf(s) ≡ \{ \[ j \in 1 \mapsto s[i] \] : i \in \text{DOMAIN} 1 \ldots \text{Len}(s) \}

If s is a string, then OneOf(s) is the set of strings formed from the individual characters of s. For example, OneOf("abc") = \{"a", "b", "c"\}.

tok(s) ≡ \{(s)\}

If s is a string, then tok(s) is the set containing only the token made from s. If S is a set of strings, then Tok(S) is the set of tokens made from elements of S.

OPERATORS FOR DEFINING LANGUAGES

Nil ≡ \{(\)\} The language containing only the “empty” sentence.

L \& M ≡ \{ s \circ t : s \in L, t \in M \} All concatenations of sentences in L and M.

L \mid M ≡ L \cup M

L^+ ≡ L \mid L \& L \mid L \& L \& L \mid \ldots

LET LL[n \in \text{Nat}] ≡ \begin{cases} L & \text{IF } n = 0 \text { THEN } L \\ LL[n-1] & \text{ELSE } LL[n-1] \mid LL[n-1] \mid L \end{cases}

IN UNION \{ LL[n] : n \in \text{Nat} \}

L^* ≡ \text{Nil} \mid L^+

L ::= M ≡ L = M

Grammar ≡ [\text{STRING} \rightarrow \text{SUBSET Seq(STRING)}]

LeastGrammar(P) ≡ The smallest grammar G such that P(G) is true.

\begin{align*}
\text{CHOSE } G & \in \text{Grammar} : \\
& \land P(G) \\
& \land \forall H \in \text{Grammar} : P(H) \Rightarrow \forall s \in \text{STRING} : G[s] \subseteq H[s]
\end{align*}

Figure 14.2: The module BNFGrammars
AtLeast4(s) ≡ Tok(\{s ◦ s ◦ s\} \& \{s\}⁺)

We now define some sets of lexemes. First is ReservedWord, the set of words that can’t be used as identifiers. (Note that BOOLEAN, TRUE, FALSE, and STRING are identifiers that are predefined.)


Here are three sets of characters—more precisely, sets of 1-character lexemes. They are the sets of letters, numbers, and characters that can appear in an identifier.

Letter ≡ OneOf(“abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ”)

Numeral ≡ OneOf(“0123456789”)

NameChar ≡ Letter \{“_”\}

We now define some sets of tokens. A Name is a token composed of letters, numbers, and _ characters that contains at least one letter. It can be used as the name of a record component or a module. An Identifier is a Name that isn’t a reserved word.

Name ≡ Tok(NameChar* \& Letter \& NameChar*)

Identifier ≡ Name \& Tok(ReservedWord)

An IdentifierOrTuple is either an identifier or a tuple of identifiers. Note that ( ) is typed as << >>.

IdentifierOrTuple ≡ Identifier \& CommaList(Identifier) \& tok(“>”)

A Number is a token representing a number. You can write the integer 63 in the following ways: 63, 63.00, \b111111 or \b11111111 (binary), \o77 or \o177 (octal), or \h3f, \h3f, \h3f, or \h3f (hexadecimal).

NumberLexeme ≡ Numeral⁺

Number ≡ Tok(NumberLexeme)

A String token represents a literal string. See Section 15.1.8 on page 195 to find out how special characters are typed in a string.

String ≡ Tok(\{“”\} \& STRING \& \{“”\})
14.1. THE SIMPLE GRAMMAR

We next define the sets of tokens that represent prefix operators (like □), infix operators (like +), and postfix operators (like prime (!)). See Figure 14.4 on page 183 to find out what symbols these ASCII strings represent.

**PrefixOp** $\triangleq$ $\text{Tok}\{\text{"-"}, \text{","}, \text{"[]"}, \text{"<>"}, \text{"DOMAIN"},$

$\text{"ENABLED"}, \text{"SUBSET"}, \text{"UNCHANGED"}, \text{"UNION"}\}$

**InfixOp** $\triangleq$

$\text{Tok}\{\text{"!"}, \text{"\#"}, \text{"\#\#"}, \text{"$"}, \text{"$$"}, \text{"\h"}, \text{"\h\h"},$

$\text{"&"}, \text{"&&"}, \text{"\{+\}"}, \text{"\{-\}"}, \text{"\{.\}"}, \text{"\{/\}"}, \text{"\{\\\}"},$\n
$\text{"\{-\}"}, \text{"\{.\}"}, \text{"\{\\\}"}, \text{"\{-\}"}, \text{"\{.\}"}, \text{"\{\\\}"},$\n
$\text{"\{-\}"}, \text{"\{.\}"}, \text{"\{\\\}"}, \text{"\{-\}"}, \text{"\{.\}"}, \text{"\{\\\}"},$\n
$\text{\"\\approx\"}, \text{\"\\asymp\"}, \text{\"\bigcirc\"}, \text{\"\bullet\"}, \text{\"\cap\"}, \text{\"\cdots\"}, \text{\"\circ\"},$\n
$\text{\"\cup\"}, \text{\"\div\"}, \text{\"\dot{\,}e\"}, \text{\"\equiv\"}, \text{\"\geq\"}, \text{\"\gg\"}, \text{\"\in\"},$\n
$\text{\"\land\"}, \text{\"\leq\"}, \text{\"\ll\"}, \text{\"\lor\"}, \text{\"\mod\"}, \text{\"\otimes\"}, \text{\"\preccurlyeq\"},$\n
$\text{\"\propto\"}, \text{\"\seq\"}, \text{\"\setminus\"}, \text{\"\cdot\circ\"}, \text{\"\sqcap\"}, \text{\"\sqcup\"}, \text{\"\sqsubset\"},$\n
$\text{\"\sqsupset\"}, \text{\"\sqsupseteq\"}, \text{\"\setminus\"}, \text{\"\subset\"}, \text{\"\subseteq\"}, \text{\"\succ\"}, \text{\"\succeq\"},$\n
$\text{\"\supseteq\"}, \text{\"\supset\"}, \text{\"\uplus\"}, \text{\"\wedge\"}, \text{\"\wedge\"}\}$

**PostfixOp** $\triangleq$ $\text{Tok}\{\text{"-"}, \text{"*"}, \text{"\#"}, \text{"\,\"}\}$

Formally, the grammar **TLAPlusGrammar** of TLA+ is the smallest grammar satisfying the BNF productions below.

**TLAPlusGrammar** $\triangleq$

$$\text{LET} \ P(G) \ \triangleq$$

Here is the BNF grammar. Terms that begin with “G.”, like **G.Module**, represent nonterminals. The terminals are sets of tokens, either defined above or described with the operators **tok** and **Tok**. The operators **AtLeast4** and **CommaList** are defined above.

$$\wedge \ G.Module ::= \text{AtLeast4(\"-\")} \ & \ \text{tok(\"MODULE\")} \ & \ Name \ & \ \text{AtLeast4(\"-\")}$$

$$\& \ (G.Unit)^* \ & \ \text{AtLeast4(\"\,\")}$$
\[ G.\ Unit \ ::= \ G.\ VariableDeclaration \]
\[ \quad | \ G.\ ConstantDeclaration \]
\[ \quad | \ (\text{Nil} \ \text{tok} \text{("LOCAL")}) \ \& \ G.\ OperatorDefinition \]
\[ \quad | \ (\text{Nil} \ \text{tok} \text{("LOCAL")}) \ \& \ G.\ FunctionDefinition \]
\[ \quad | \ (\text{Nil} \ \text{tok} \text{("LOCAL")}) \ \& \ G.\ Instance \]
\[ \quad | \ (\text{Nil} \ \text{tok} \text{("LOCAL")}) \ \& \ G.\ ModuleDefinition \]
\[ \quad | \ G.\ Assumption \]
\[ \quad | \ G.\ Theorem \]
\[ \quad | \ G.\ Module \]
\[ \quad | \ \text{AtLeastA("\_"}) \]

\[ G.\ VariableDeclaration \ ::= \]
\[ \quad \text{Tok} \{\text{"VARIABLE"}, \text{"VARIABLES"}\} \ \& \ \text{CommaList}(\text{Identifier}) \]

\[ G.\ ConstantDeclaration \ ::= \]
\[ \quad \text{Tok} \{\text{"CONSTANT"}, \text{"CONSTANTS"}\} \ \& \ \text{CommaList}(G.\ OpDecl) \]

\[ G.\ OpDecl \ ::= \ \text{Identifier} \]
\[ \quad | \ \text{Identifier} \ \& \ \text{tok} \text{("\_")} \ \& \ \text{CommaList}(\text{tok} \text{("\_")} \ \& \ \text{tok} \text{("\_")}) \]
\[ \quad | \ \text{PrefixOp} \ \& \ \text{tok} \text{("\_")} \]
\[ \quad | \ \text{tok} \text{("\_")} \ \& \ \text{InfixOp} \ \& \ \text{tok} \text{("\_")} \]
\[ \quad | \ \text{tok} \text{("\_")} \ \& \ \text{PostfixOp} \]

\[ G.\ OperatorDefinition \ ::= \]
\[ \quad \{ \text{G.\ NonFixLHS} \]
\[ \quad \quad | \ \text{PrefixOp} \ \& \ \text{Identifier} \]
\[ \quad \quad | \ \text{Identifier} \ \& \ \text{InfixOp} \ \& \ \text{Identifier} \]
\[ \quad \quad | \ \text{Identifier} \ \& \ \text{PostfixOp} \]
\[ \quad \& \ \text{tok} \text{("\_")} \]
\[ \quad | \ \text{G.\ Expression} \]

\[ G.\ NonFixLHS \ ::= \]
\[ \quad \text{Identifier} \]
\[ \quad \& \ \text{Nil} \]
\[ \quad \ | \ \text{tok} \text{("\_")} \ \& \ \text{CommaList}(\text{Identifier} \ | \ G.\ OpDecl) \ \& \ \text{tok} \text{("\_")} \]

\[ G.\ FunctionDefinition \ ::= \]
\[ \quad \text{Identifier} \]
\[ \quad \& \ \text{tok} \text{("\_")} \ \& \ \text{CommaList}(G.\ QuantifierBound) \ \& \ \text{tok} \text{("\_")} \]
\[ \quad \& \ \text{tok} \text{("\_")} \]
\[ \quad | \ \text{G.\ Expression} \]
14.1. THE SIMPLE GRAMMAR

\[ G.\text{QuantifierBound} ::= ( \text{IdentifierOrTuple} | \text{CommaList(Identifier)} ) \]
\[ G.\text{QuantifierBound} \& \text{tok("\in")} \]
\[ G.\text{QuantifierBound} \& G.\text{Expression} \]

\[ G.\text{Instance} ::= \text{tok("INSTANCE")} \]
\[ G.\text{Instance} \& \text{Name} \]
\[ G.\text{Instance} \& ( \text{Nil} \& \text{tok("WITH")} \& \text{CommaList(G.Substitution)}) \]

\[ G.\text{Substitution} ::= ( \text{Identifier} \& \text{PrefixOp} \& \text{InfixOp} \& \text{PostfixOp}) \]
\[ G.\text{Substitution} \& \text{tok("<-")} \]
\[ G.\text{Substitution} \& G.\text{Argument} \]

\[ G.\text{Argument} ::= G.\text{Expression} \]
\[ G.\text{Argument} \& G.\text{GeneralPrefixOp} \]
\[ G.\text{Argument} \& G.\text{GeneralInfixOp} \]
\[ G.\text{Argument} \& G.\text{GeneralPostfixOp} \]

\[ G.\text{InstancePrefix} ::= \text{tok("!") \& CommaList(G.Expression) \& tok("!”)} \]
\[ G.\text{InstancePrefix} \& G.\text{Instance} \]

\[ G.\text{ModuleDefinition} ::= G.\text{NonFixLHS} \& \text{tok("==")} \& G.\text{Instance} \]

\[ G.\text{Assumption} ::= \text{tok(\{"ASSUME", "ASSUMPTION", "AXIOM"\})} \& G.\text{Expression} \]

\[ G.\text{Theorem} ::= \text{tok("THEOREM")} \& G.\text{Expression} \]

The comments give examples of each of the different types of expression.

\[ G.\text{Expression} ::= \]
\[ G.\text{GeneralIdentifier} \; A(x+7)!B!1d \]
\[ | G.\text{GeneralIdentifier} \& \text{tok("") \& CommaList(G.Argument) \& tok(“")} \; A!Op(x+1, y) \]
\[ | G.\text{GeneralPrefixOp} \& G.\text{Expression} \; \text{DOMAIN}(S \cup T) \]
\[ a + b \]
\[ (x + 1) \]
\[ \forall x \in S, (y, z) \in T \to F(x, y, z) \]
\[ \exists x, y : x + y > 0 \]
\[ \text{choose (} x, y \text{) } \in S : F(x, y) \]
\[ \{ x \in \text{Nat} : x > 0 \} \]
\[ \{ F(x, y, z) : x, y \in S, z \in T \} \]
\[ [i, j \in S, (p, q) \in T \to F(i, j, p, q)] \]
\[ ([S \cup T] \to U] \]
\[ [a \to x + 1, b \to y] \]
\[ [a : \text{Nat}, b : S] \]
14.1. THE SIMPLE GRAMMAR

tok("["") \text{ [f except ![1,x], r = 4, ![2,y]] = e] \& G.Expression
& tok("EXCEPT")
& CommaList(tok("!")
& (tok(“.”) \& Name
& | tok("["") \& CommaList(G.Expression) \& tok("]") +
& tok("=") \& G.Expression)
& tok("]")
& tok("<") \& CommaList(G.Expression) \& tok(">>") (1,2,3)
G.Expression \& (Tok(\{"\X", "\times"\}) \& G.Expression) \& Nat \times Nat \times Real
& tok("["]) \& G.Expression \& tok(".")] \& G.Expression [\text{Next}](x,y)
& tok("<<") \& G<Expression \& tok(">>") \& G.Expression \& tok("") \& G.Expression \& tok(""")
Tok(\{"WF_\_", "SF_\_"\}) \& G.Expression \& tok(""") \& G.Expression \& tok(""")
\& tok("fas") \& G.Expression \& tok("THEN") \& G.Expression \& tok("ELSE") \& G.Expression
\& tok("CASE")
& (let CaseArm \triangleq G.Expression \& tok("->") \& G.Expression
\& IN CaseArm \& (tok("["]) \& CaseArm)* )
& (Nil
\& (tok("["]) \& tok("OTHER") \& tok("->") \& tok(""") \& G.Expression))
& tok("LET")
& (let x \triangleq y + 1 \& f[t \in Nat] \triangleq t^2
\& IN x + f[y]
\& L.OperatorDefinition
\& L.FunctionDefinition
\& L.ModuleDefinition)*
& tok("IN")
& G.Expression
\& (tok("/")) \& G.Expression
\& G.Expression
\& (tok(\"/\") \& G.Expression + \& x = 1
\& y = 2
\& (tok(\"/\") \& G.Expression + \& x = 1
\& y = 2
\& Number 09001
14.2 The Complete Grammar

We now complete our explanation of the syntax of TLA\(^+\) by giving the details that are not described by the BNF grammar in the previous section. Section 14.2.1 gives the precedence rules, Section 14.2.2 gives the alignment rules for conjunction and disjunction lists, and Section 14.2.3 describes comments. Section 14.2.4 briefly discusses the syntax of temporal formulas. Finally, for completeness, Section 14.2.5 explains the handling of two anomalous cases that you’re unlikely ever to encounter.

14.2.1 Precedence and Associativity

The expression \(a + b * c\) is interpreted as \(a + (b * c)\) rather than \((a + b) * c\). This convention is described by saying that the operator \(*\) has higher precedence than the operator \(+\). In general, operators with higher precedence are applied before operators of lower precedence. This applies to prefix operators (like SUBSET) and postfix operators (like ‘\() as well as to infix operators like \(+\) and \(*\). Thus, \(a + b’\) is interpreted as \(a + (b’\) because ‘\) has higher precedence than \(+\). Application order can also be determined by associativity. The expression \(a - b - c\) is interpreted as \((a - b) - c\) because \(-\) is a left-associative infix operator.

In TLA\(^+\), the precedence of an operator is a range of numbers, like 7–13. The operator \(\$\) has higher precedence than the operator \(\Rightarrow\) because the precedence of \(\$\) is 7–12, and this entire range is greater than the precedence range of \(\Rightarrow\), which is 6–6. An expression is illegal (syntactically incorrect) if the order of application of two operators is not determined because their precedence ranges overlap and they are not two instances of an associative infix operator. For example, the expression \(a + b * c’ \% d\) is illegal for the following reason. The precedence range of ‘\) is higher than that of \(*\) and \%, and the precedence range of \(*\) is higher than that of \(+\) and \%, so this expression is interpreted as \(a + (b * (c’)) \% d\). However, the precedences of \(+\) (9–9) and \% (9–10) overlap, so the expression is illegal.

TLA\(^+\) embodies the philosophy that it’s better to require parentheses than to allow expressions that could be easily be misinterpreted. Thus, \(*\) and \(/\) have overlapping precedence, making an expression like \(a/b * c\) illegal. (This also makes \(a * b/c\) illegal, even though \((a * b)/c\) and \(a * (b/c)\) happen to be equal when \(*\) and \(/\) have their usual definitions.) Unconventional operators like \(\$\) have
wide precedence ranges for safety. But, even when the precedence rules imply that parentheses aren’t needed, it’s often a good idea to use them anyway if you think there’s any chance that a reader might not understand how an Expression is interpreted.

Figure 14.3 on the next page gives the precedence ranges of all operators and tells which infix operators are left associative. (No TLA operators are right associative.)

There are two precedence rules not covered by the operator precedence ranges. The first is that function application has higher precedence than any prefix or infix operator. Thus, \( a + b[c] \) is interpreted as \( a + (b[c]) \). The second rule involves the following TLA expression-making constructs that have no explicit right-hand terminator: \texttt{CHOOSE}, \texttt{IF/THEN/ELSE}, \texttt{CASE}, and \texttt{LET/IN} constructs, and quantifiers. These constructs are treated as prefix operators with the lowest possible precedence, so an expression made with one of them extends as far as possible. More precisely, the expression is ended only by one of the following:

- The beginning of the next module unit. (Module units are produced by the \texttt{Unit} nonterminal in the BNF grammar of Section 14.1.2; they include definition and declaration statements.)
- A right delimiter whose matching left delimiter occurs before the beginning of the construct. Delimiter pairs are \( (,) \), \( [ ] \), \( \{ \} \), and \( (\) \).
- Any of the following lexemes: \texttt{THEN}, \texttt{ELSE}, \texttt{IN}, \texttt{comma}, \texttt{colon}, \texttt{→}, and \( √ \) (when it is the separator in a \texttt{CASE} statement, and not the prefix temporal operator).
- Any symbol not to the right of the \( ∧ \) or \( ∨ \) prefixing a conjunction or disjunction list element containing the construct. (See Section 14.2.2 below.)

Here is how some expressions are interpreted under this rule:

\[
\begin{align*}
\text{IF } x > 0 \text{ THEN } y + 1 & \quad \text{IF } x > 0 \text{ THEN } y + 1 \\
\text{ELSE } y - 1 & \quad \text{ELSE } (y - 1 + 2)
\end{align*}
\]

\[
\forall x \in S : P(x) \quad \text{means} \quad \forall x \in S : (P(x) \lor Q)
\]

As these examples show, indentation is ignored—except in conjunction and disjunction lists, discussed below. The absence of a terminating lexeme (an \texttt{END}) for an \texttt{IF/THEN/ELSE} or \texttt{CASE} construct usually makes an expression less cluttered, but sometimes it does require you to add parentheses instead.
### Prefix Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Enabled</th>
<th>Subset</th>
<th>Union</th>
<th>Unchanged</th>
<th>Domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Rightarrow$</td>
<td>1-1</td>
<td></td>
<td></td>
<td>3-14</td>
<td></td>
</tr>
<tr>
<td>$\nRightarrow$</td>
<td>2-2</td>
<td></td>
<td></td>
<td>4-14</td>
<td></td>
</tr>
<tr>
<td>$\Leftarrow$</td>
<td>2-2</td>
<td></td>
<td></td>
<td>5-5</td>
<td></td>
</tr>
<tr>
<td>$\Rightarrow$</td>
<td></td>
<td></td>
<td></td>
<td>7-7</td>
<td></td>
</tr>
<tr>
<td>$\Leftarrow$</td>
<td></td>
<td></td>
<td></td>
<td>8-8</td>
<td></td>
</tr>
<tr>
<td>$\Leftarrow$</td>
<td></td>
<td></td>
<td></td>
<td>9-9</td>
<td></td>
</tr>
<tr>
<td>$\Leftarrow$</td>
<td></td>
<td></td>
<td></td>
<td>10-10</td>
<td></td>
</tr>
<tr>
<td>$\Leftarrow$</td>
<td></td>
<td></td>
<td></td>
<td>11-11</td>
<td></td>
</tr>
</tbody>
</table>

### Infix Operators

#### (1) Action composition ($\cdot$).

<table>
<thead>
<tr>
<th>Operator</th>
<th>Enabled</th>
<th>Subset</th>
<th>Union</th>
<th>Unchanged</th>
<th>Domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\gg$</td>
<td>7-7</td>
<td></td>
<td></td>
<td>3-14</td>
<td></td>
</tr>
<tr>
<td>$\ggg$</td>
<td>7-7</td>
<td></td>
<td></td>
<td>7-7</td>
<td></td>
</tr>
<tr>
<td>$\gggg$</td>
<td></td>
<td></td>
<td></td>
<td>10-10</td>
<td></td>
</tr>
<tr>
<td>$\ggggg$</td>
<td></td>
<td></td>
<td></td>
<td>12-12</td>
<td></td>
</tr>
<tr>
<td>$\gggggg$</td>
<td></td>
<td></td>
<td></td>
<td>14-14</td>
<td></td>
</tr>
<tr>
<td>$\ggggggg$</td>
<td></td>
<td></td>
<td></td>
<td>16-16</td>
<td></td>
</tr>
</tbody>
</table>

#### (2) Record component (period).

<table>
<thead>
<tr>
<th>Operator</th>
<th>Enabled</th>
<th>Subset</th>
<th>Union</th>
<th>Unchanged</th>
<th>Domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\ggg$</td>
<td>7-7</td>
<td></td>
<td></td>
<td>4-14</td>
<td></td>
</tr>
<tr>
<td>$\gggg$</td>
<td>7-7</td>
<td></td>
<td></td>
<td>4-14</td>
<td></td>
</tr>
<tr>
<td>$\ggggg$</td>
<td></td>
<td></td>
<td></td>
<td>7-7</td>
<td></td>
</tr>
<tr>
<td>$\gggggg$</td>
<td></td>
<td></td>
<td></td>
<td>7-7</td>
<td></td>
</tr>
<tr>
<td>$\ggggggg$</td>
<td></td>
<td></td>
<td></td>
<td>7-7</td>
<td></td>
</tr>
</tbody>
</table>

### Postfix Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Enabled</th>
<th>Subset</th>
<th>Union</th>
<th>Unchanged</th>
<th>Domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\ggg$</td>
<td>7-7</td>
<td></td>
<td></td>
<td>14-14</td>
<td></td>
</tr>
<tr>
<td>$\gggg$</td>
<td>7-7</td>
<td></td>
<td></td>
<td>14-14</td>
<td></td>
</tr>
<tr>
<td>$\ggggg$</td>
<td></td>
<td></td>
<td></td>
<td>14-14</td>
<td></td>
</tr>
</tbody>
</table>

Figure 14.3: The precedence ranges of operators. Left-associative operators are indicated by (a).
14.2.2 Alignment

The most novel aspect of TLA\(^+\) syntax is the aligned conjunction and disjunction lists. If you write such a list in a straightforward manner, then it will mean what you expect it to. However, you might wind up doing something weird through a typing error. So, it’s a good idea to know what the exact syntax rules are for these lists. I give the rules here for conjunction lists; the rules for disjunction lists are the same.

A conjunction list is an expression that begins with \(\land\), which is typed as \(\land\). Let \(c\) be the column in which the \(\land\) occurs. (The first character in a line is in column 1.) The conjunction list consists of a sequence of conjuncts, each beginning with a \(\land\). A conjunct is ended by any one of the following that occurs after the \(\land\):

1. Another \(\land\) whose \(\land\) character is in column \(c\) and is the first nonspace character on the line.
2. Any nonspace character in column \(c\) or a column to the left of column \(c\).
3. A right delimiter whose matching left delimiter occurs before the beginning of the conjunction list. Delimiter pairs are \((\ ), \[\], \{\}, and \(\}\).
4. The beginning of the next module unit. (Module units are produced by the \textit{Unit} nonterminal in the BNF grammar of Section 14.1.2; they include definition and declaration statements.)

In case 1, the \(\land\) begins the next conjunct in the same conjunction list. In the other three cases, the end of the conjunct is the end of the entire conjunction list. In all cases, the character ending the conjunct does not belong to the conjunct. With these rules, indentation properly delimits expressions in a conjunction list—for example:

\[
\land \text{ IF } e \text{ THEN } P \quad \text{\& (IF } e \text{ THEN } P \\
\land \text{ ELSE } Q \quad \text{\& ELSE } Q \\
\land \text{ R } \quad \text{\& R}
\]

It’s best to indent each conjunction completely to the right of its \(\land\) symbol. These examples illustrate precisely what happens if you don’t:

\[
\land \text{ x' } \quad \text{\& } x' = y \\
\text{\& y' = x } \quad \text{\& y' = x } \quad \text{\& (y' = x)}
\]

In the second example, the \(\land \text{ x' }\) is interpreted as a conjunction list containing only one conjunct, and the second \(\land\) is interpreted as an infix operator.

You can’t use parentheses to circumvent the indentation rules. For example, this is illegal:
\( \backslash \ (x') = y \)
\( \backslash \ y' = x \)

The rules imply that the first \( \backslash \) begins a conjunction list that is ended before the \( \ast \). That conjunction list is therefore \( \land (x') \), which has an unmatched left parenthesis.

The conjunction/disjunction list notation is quite robust. It is easy to screw up the alignment by typing one space too few or too many—especially when each conjunct is a large formula. However, the formula is still likely to mean what you intended. Here’s an example of what happens if you misalign a conjunct:

\[
\begin{align*}
\backslash \ A & \quad \land \ A \\
\land \ B & \quad \text{means} \quad \land \ B \\
\land \ C & \quad \land \ C
\end{align*}
\]

(The last two \( \land \) symbols are interpreted as infix operators.) While not interpreted the way you expected, this formula is equivalent to \( A \land B \land C \), which is what you meant in the first place.

Most keyboards contain one key that is the source of a lot of trouble: the tab key (sometimes marked on the keyboard with a right arrow). On my computer screen, I can produce

\[
A = \quad \land \ x' = 1 \\
\land \ y' = 2
\]

by beginning the second line with eight space characters and the third with one tab character. In this case, it is unspecified whether or not the two \( / \) characters occur in the same column. Tab characters are an anachronism left over from the days of typewriters and computers with memory capacity measured in kilobytes. I strongly advise you never to use them. But, if you insist on using them, here are the rules:

- A tab character is considered to be equivalent to one or more space characters, so it occupies one or more columns.
- Identical sequences of space and tab characters that occur at the beginning of a line occupy the same number of columns.

There are no other guarantees if you use tab characters.

## 14.2.3 Comments

Comments are described in Section 3.5 on page 32. A comment may appear between any two lexemes in a specification. There are two types of comments:
14.2. THE COMPLETE GRAMMAR

- A delimited comment is a string of the form “(* \textit{\textcolor{red}{s}} * “)”, where \textit{\textcolor{red}{s}} is any string not containing the substring “(* “).
- An end-of-line comment is a string of the form “* \textit{\textcolor{red}{s}} \textcolor{red}{\textbackslash hlf} “, where \textit{\textcolor{red}{s}} is any string not containing an end-of-line character (LF).

I like to write comments as shown here:

\begin{verbatim}
BufRcv == /
\textcolor{red}{/\textit{InChan!Rcv (\\textcolor{red}{******\\textit{\textcolor{red}{}}}}} \textcolor{3}
\textcolor{red}{/\textit{q' = Append(q, in.val)} (* Receive message from channel *) \textcolor{3}
\textcolor{red}{/\textit{out}} (* in and append to tail of q. *) \textcolor{3}
\textcolor{red}{\textcolor{3}{******\\textit{\textcolor{red}{}}))}
\end{verbatim}

Grammatically, this piece of specification has four distinct comments, the first and last consisting of the same string (****...****). But a person reading it would regard these as forming a single comment, spread over four lines. This kind of commenting convention is not part of the TLA\textsuperscript{+} language, but it might be supported by tools such as a pretty-printer.

14.2.4 Temporal Formulas

The BNF grammar treats $\Box$ and $\Diamond$ simply as prefix operators. However, as explained in Section 8.1 (page 85), the syntax of temporal formulas places restrictions on their use. For example, $\Box(x' = x + 1)$ is not a legal formula. It’s not hard to write a BNF grammar that specifies legal temporal formulas made from the temporal operators and ordinary Boolean operators like $\neg$ and $\land$. However, such a BNF grammar won’t tell you which of these two expressions is legal:

\begin{verbatim}
LET \textcolor{red}{F(P, Q) \triangleq P \land \Box Q) \textcolor{3}
IN \textcolor{red}{F(x = 1, x = y + 1)} \textcolor{3}
\textcolor{red}{\textcolor{3}{******\\textit{\textcolor{red}{}})}}
\end{verbatim}

\begin{verbatim}
LET F(P, Q) \triangleq P \land \Box Q)
IN F(x = 1, x' = y + 1)
\end{verbatim}

(The first is legal; the second isn’t.) The precise rules for determining the syntactic correctness of a temporal formula involve first replacing all defined operators by their definitions, using the procedure described in Section 16.4 below. I won’t bother specifying these rules.

In practice, temporal operators are not used very much in TLA\textsuperscript{+} specifications, and one rarely writes definitions of new ones such as

\begin{verbatim}
F(P, Q) \triangleq P \land \Box Q)
\end{verbatim}

The syntactic rules for expressions involving such operators are of academic interest only.
14.2.5 Two Anomalies

There are two sources of ambiguity in the grammar of TLA\(^+\) that you are unlikely to encounter and that have \textit{ad hoc} resolutions. The least unlikely of these arises from the use of \(\neg\) as both an infix operator (as in \(2 + 2\)) and a prefix operator (as in \(2 + -2\)). This poses no problem when \(\neg\) is used in an ordinary expression. However, there are two places in which an operator can appear by itself:

- As the argument of a higher-order operator, as in \(HOp(+, -)\).

- In an \texttt{instance} substitution, such as

  \[
  \text{instance } M \text{ with } \text{Plus} \leftarrow +, \text{Minus} \leftarrow -
  \]

In both these cases, the symbol \(\neg\) is interpreted as the infix operator. You must type \(-\) to denote the prefix operator. You also have to type \(-\) if you should ever want to define the prefix \(\neg\) operator, as in:

\[
\neg a \triangleq \text{UMinus}(a)
\]

Remember that, in ordinary expressions, you just type \(-\) as usual for both operators.

The second source of ambiguity in the TLA\(^+\) syntax is an unlikely expression of the form \(\{x \in S : y \in T\}\), which might be taken to mean either of the following:

\[
\begin{align*}
\text{let } & p \triangleq y \in T \text{ in } \{x \in S : p\} \\
\text{let } & p \triangleq x \in S \text{ in } \{p : y \in T\}
\end{align*}
\]

It is interpreted as the first formula.

14.3 What You Type

The grammar in Section 14.1.2 describes the ASCII syntax for TLA\(^+\) specifications. Typeset versions of specifications appear in this book. For example, the grammar lists the infix operator \(\prec\), but that operator is printed in specifications as \(\ll\). Figure 14.4 on the next page gives the correspondence between the ASCII and typeset versions of all TLA\(^+\) symbols for which the correspondence may not be obvious.

Finally, Figure 14.5 on page 184 lists all the user-definable infix, postfix, and prefix operator symbols of TLA\(^+\). It also indicates which of them are defined by the standard modules. This is a good place to look when choosing notation for your specification.
\^ or \land \quad \lor or \lor \quad \Rightarrow \quad =>
\neg or \neg \quad \equiv \quad <= or \equiv \quad \equiv \quad ==
\{ or \{ \quad \} or \} \quad \square \quad []

\ll or <= \quad \ll or <= \quad \sim or \sim \quad ~> or ~>
\ll or <= \quad \ll or <= \quad \sim or \sim \quad ~> or ~>

\ll \quad \ll \quad \ll
\preccurlyeq \quad \preccurlyeq \quad \\div \quad \\div
\preceq \quad \preceq \quad \preceq
\subseteq \quad \subseteq \quad \subseteq
\subset \quad \subset \quad \subset
\preceq \quad \preceq \quad \preceq
\sqsubset \quad \sqsubset \quad \sqsubset
\subseteq \quad \subseteq \quad \subseteq
\mathcal{A} \quad \mathcal{A} \quad \mathcal{A}

\cap or \cap \quad \cup or \cup \quad \cap or \cap \quad \cup or \cup
\mathcal{A} \quad \mathcal{A} \quad \mathcal{A}
\mathcal{A} \quad \mathcal{A} \quad \mathcal{A}

\cap or \cap \quad \cup or \cup \quad \cap or \cap \quad \cup or \cup
\mathcal{A} \quad \mathcal{A} \quad \mathcal{A}
\mathcal{A} \quad \mathcal{A} \quad \mathcal{A}

\text{(1)} \ s \text{ is a sequence of characters. See Section 15.1.8 on page 195.}
\text{(2)} \ x \text{ and } y \text{ are any expressions.}
\text{(3)} \ a \text{ sequence of four or more } \text{- or } = \text{ characters.}

Figure 14.4: The \textsc{Asch} representations of typeset symbols.
CHAPTER 14. THE SYNTAX OF TLA+

Infix Operators

\begin{itemize}
  \item $+$ (1)
  \item $-$ (1)
  \item $\ast$ (1)
  \item $/$ (2)
  \item $\circ$ (3)
  \item $++$
  \item $-\div$ (1)
  \item $\%$ (1)
  \item $\wedge$ (1.4)
  \item $.$ (1)
  \item $\ldots$
  \item $--$
  \item $\oplus$ (5)
  \item $\otimes$
  \item $\ominus$
  \item $\otimes$
  \item $**$
  \item $\triangleleft$ (1)
  \item $\triangleright$ (1)
  \item $\triangleleft$ (1)
  \item $\triangleright$ (1)
  \item $\equiv$
  \item $\triangleright$
  \item $:$
  \item $:$
  \item $\&$
  \item $\&$
  \item $\&$
  \item $\&$
  \item $\circ$
  \item $\circ$
  \item $\circ$
  \item $\circ$
  \item $\circ$
  \item $\circ$
  \item $\circ$
  \item $\circ$
  \item $\circ$
  \item $\circ$
  \item $\circ$
  \item $\circ$
  \item $\circ$
  \item $\circ$
  \item $\circ$
  \item $\circ$
  \item $\circ$
  \item $\circ$
  \item $\circ$
  \item $\circ$
  \item $\circ$
  \item $\circ$
  \item $\circ$
  \item $\circ$
  \item $\circ$
  \item $\circ$
  \item $\circ$
  \item $\circ$
  \item $\circ$
  \item $\circ$
  \item $\circ$
  \item $\circ$
  \item $\circ$
  \item $\circ$
  \item $\circ$
  \item $\circ$
  \item $\circ$
  \item $\circ$
  \item $\circ$
  \item $\circ$
  \item $\circ$
  \item $\circ$
  \item $\circ$
  \item $\circ$
  \item $\circ$
  \item $\circ$
  \item $\circ$
  \item $\circ$
  \item $\circ$
  \item $\circ$
  \item $\circ$
  \item $\circ$
  \item $\circ$
  \item $\circ$
  \item $\circ$
  \item $\circ$
  \item $\circ$
  \item $\circ$
  \item $\circ$
  \item $\circ$
  \item $\circ$
  \item $\circ$
  \item $\circ$
  \item $\circ$
  \item $\circ$
  \item $\circ$
  \item $\circ$
  \item $\circ$
  \item $\circ$
  \item $\circ$
  \item $\circ$
  \item $\circ$
  \item $\circ$
  \item $\circ$
  \item $\circ$
  \item $\circ$
  \item $\circ$
  \item $\circ$
  \item $\circ$
  \item $\circ$
  \item $\circ$
  \item $\circ$
  \item $\circ$
  \item $\circ$
  \item $\circ$
  \item $\circ$
  \item $\circ$
  \item $\circ$
  \item $\circ$
  \item $\circ$
  \item $\circ$
  \item $\circ$
  \item $\circ$
  \item $\circ$
  \item $\circ$
  \item $\circ$
  \item $\circ$
  \item $\circ$
  \item $\circ$
  \item $\circ$
  \item $\circ$
  \item $\circ$
  \item $\circ$
  \item $\circ$\end{itemize}

Postfix Operators (6)

$^+$ $^*$ $^#$

Prefix Operator

$-$ (7)

\begin{enumerate}
  \item Defined by the \textit{Naturals}, \textit{Integers}, and \textit{Reals} modules.
  \item Defined by the \textit{Reals} module.
  \item Defined by the \textit{Sequences} module.
  \item $x^y$ is printed as $x^y$.
  \item Defined by the \textit{Bags} module.
  \item $e^+$ is printed as $e^+$, and similarly for $^*$ and $^#$.
  \item Defined by the \textit{Integers} and \textit{Reals} modules.
\end{enumerate}

Figure 14.5: User-definable operator symbols.
Chapter 15

The Operators of TLA\(^+\)

This section describes the built-in operators of TLA\(^+\). Most of these operators have been described in Part I. So, this section will serve as a reference manual to describe the fine points, assuming that you understand the basic meaning of the operators. It provides an informal mathematical semantics of the operators.

15.1  Constant Operators

Constant operators are the operators of ordinary mathematics, having nothing to do with TLA or temporal logic. The constant operators of TLA\(^+\) are listed in Figure 15.1 on the next page and Figure 15.2 on page 187. Their meanings are defined below.

15.1.1  Boolean Operators

Propositional logic operators are described in Section 1.1. TLA\(^+\) provides the usual ones:

\[ \land \quad \lor \quad \neg \quad \Rightarrow \text{ (implication)} \quad \equiv \quad \text{TRUE} \quad \text{FALSE} \]

The identifier BOOLEAN is defined to be the set \{TRUE, FALSE\}. Conjunctions and disjunctions can be written as aligned lists,

\[
\begin{align*}
\land \ p_1 & \quad \lor \ p_1 \\
\vdots & \quad \Delta \quad p_1 \land \ldots \land p_n \\
\land \ p_n & \quad \lor \ p_n \\
\end{align*}
\]

Predicate logic operators are described in Section 1.3. The unbounded operators have the general forms:

\[
\forall x_1, \ldots, x_n : p \quad \exists x_1, \ldots, x_n : p
\]
CHAPTER 15. THE OPERATORS OF TLA+

Logic
\begin{align*}
\land \land \neg \Rightarrow & \equiv \\
\text{TRUE} & \quad \text{FALSE} \quad \text{BOOLEAN} \quad [\text{the set \{TRUE, FALSE\}]}
\end{align*}
\begin{align*}
\forall x : p & \quad \exists x : p \quad \forall x \in S : p \quad (1) \quad \exists x \in S : p \quad (1) \\
\text{CHOOSE } x : p \quad [\text{An } x \text{ satisfying } p] & \quad \text{CHOOSE } x \in S : p \quad [\text{An } x \text{ in } S \text{ satisfying } p]
\end{align*}

Sets
\begin{align*}
\neq & \quad \in \quad \not\in \cup \quad \cap \quad \subseteq \quad \setminus \quad [\text{set difference}] \\
\{e_1, \ldots, e_n\} & \quad [\text{Set consisting of elements } e_i] \\
\{x \in S : p\} & \quad [\text{Set of elements } x \text{ in } S \text{ satisfying } p] \\
\{e : x \in S\} & \quad [\text{Set of elements } e \text{ such that } x \text{ in } S] \\
\text{SUBSET } S & \quad [\text{Set of subsets of } S] \\
\text{UNION } S & \quad [\text{Union of all elements of } S]
\end{align*}

Functions
\begin{align*}
f[e] & \quad [\text{Function application}] \\
\text{DOMAIN } f & \quad [\text{Domain of function } f] \\
[x \in S \mapsto e] & \quad [\text{Function } f \text{ such that } f[x] = e \text{ for } x \in S] \\
[S \rightarrow T] & \quad [\text{Set of functions } f \text{ with } f[x] \in T \text{ for } x \in S] \\
f \text{ EXCEPT } ![e_1] = e_2 & \quad (3) \quad [\text{Function } \hat{f} \text{ equal to } f \text{ except } \hat{f}[e_1] = e_2]
\end{align*}

Records
\begin{align*}
e.h & \quad [\text{The } h\text{-component of record } e] \\
[h_1 \mapsto e_1, \ldots, h_n \mapsto e_n] & \quad [\text{The record whose } h_i \text{ component is } e_i] \\
[h_1 : S_1, \ldots, h_n : S_n] & \quad [\text{Set of all records with } h_i \text{ component in } S_i] \\
r \text{ EXCEPT } ![.h = e] & \quad (3) \quad [\text{Record } \hat{r} \text{ equal to } r \text{ except } \hat{r}.h = e]
\end{align*}

Tuples
\begin{align*}
e[i] & \quad [\text{The } i\text{-th component of tuple } e] \\
\langle e_1, \ldots, e_n \rangle & \quad [\text{The } n\text{-tuple whose } i\text{-th component is } e_i] \\
S_1 \times \ldots \times S_n & \quad [\text{The set of all } n\text{-tuples with } i\text{-th component in } S_i]
\end{align*}

Strings and Numbers
\begin{align*}
"e_1 \ldots e_n" & \quad [\text{A literal string of } n \text{ characters}] \\
\text{STRING} & \quad [\text{The set of all strings}] \\
d_1 \ldots d_n \quad d_1 \ldots d_n . d_{n+1} \ldots d_m & \quad [\text{Numbers (where the } d_i \text{ are digits)}]
\end{align*}

(1) \(x \in S\) may be replaced by a comma-separated list of items \(v \in S\), where \(v\) is either a comma-separated list or a tuple of identifiers.

(2) \(x\) may be an identifier or tuple of identifiers.

(3) \![e_1]\) or \![.h\) may be replaced by a comma separated list of items \!a_1 \ldots a_n, where each \(a_i\) is \([e_i]\) or \(.h_i\).

Figure 15.1: The constant operators.
15.1. CONSTANT OPERATORS

\[
\begin{align*}
\text{IF } p \text{ THEN } e_1 \text{ ELSE } e_2 & \quad [e_1 \text{ if } p \text{ true}, \text{ else } e_2] \\
\text{CASE } p_1 \rightarrow e_1 & \ldots \rightarrow p_n \rightarrow e_n & \quad [\text{Some } e_i \text{ such that } p_i \text{ true}] \\
\text{CASE } p_1 \rightarrow e_1 & \ldots \rightarrow p_n \rightarrow e_n \rightarrow \text{OTHER } \rightarrow e & \quad [\text{Some } e_i \text{ such that } p_i \text{ true}, \\
& \quad \text{or } e \text{ if all } p_i \text{ are false}] \\
\text{LET } d_1 \triangleq e_1 \ldots d_n \triangleq e_n \text{ IN } e & \quad [e \text{ in the context of the definitions}] \\
\wedge p_1 & \quad [\text{the conjunction } p_1 \wedge \ldots \wedge p_n] \\
\vee p_1 & \quad [\text{the disjunction } p_1 \lor \ldots \lor p_n] \\
\vdots & \\
\wedge p_n & \quad \vee p_n
\end{align*}
\]

Figure 15.2: Miscellaneous constructs.

where each \( x_i \) is an identifier. They are defined in terms of quantification over a single variable by:

\[
\forall x_1, \ldots, x_n : p \triangleq \forall x_1 : (\forall x_2 : (\ldots \forall x_n : p) \ldots)
\]

and similarly for \( \exists \). The bounded operators have the general forms:

\[
\forall y_1 \in S_1, \ldots, y_n \in S_n : p \quad \exists y_1 \in S_1, \ldots, y_n \in S_n : p
\]

where each \( y_i \) has the form \( x_1, \ldots, x_k \) or \( \langle x_1, \ldots, x_k \rangle \), and each \( x_j \) is an identifier. The general forms of \( \forall \) are defined by

\[
\forall y_1 \in S_1, \ldots, y_n \in S_n : p \triangleq \forall y_1 \in S_1 : \ldots : \forall y_n \in S_n : p \\
\forall x_1, \ldots, x_k \in S : p \triangleq \forall x_1 \in S : \ldots : \forall x_k \in S : p \\
\forall (x_1, \ldots, x_k) \in S : p \triangleq \forall x_1, \ldots, x_k : (\langle x_1, \ldots, x_k \rangle \in S) \Rightarrow p
\]

In these expressions, \( S \) and the \( S_i \) lie outside the scope of the quantifier’s bound identifiers. The definitions for \( \exists \) are similar. In particular:

\[
\exists (x_1, \ldots, x_k) \in S : p \triangleq \exists x_1, \ldots, x_k : (\langle x_1, \ldots, x_k \rangle \in S) \land p
\]

See Section 15.1.7 for further details about tuples.

The \textsc{choose} operator is discussed in Section 6.6. A formal exposition of \textsc{choose} is given by Leisenring [4], who calls it by its common mathematical name of Hilbert’s \( \varepsilon \). The general forms are:

\[
\textsc{choose } x : p \quad \textsc{choose } x \in S : p
\]

where \( x \) is an identifier or a tuple \( \langle x_1, \ldots, x_n \rangle \) of identifiers. In the bounded form, \( S \) lies outside the scope of the bound identifier \( x \). The bounded form is defined by:

\[
\textsc{choose } x \in S : p \triangleq \textsc{choose } x : (x \in S) \land p
\]
When \( x \) is an identifier, the unbounded \texttt{CHOOSE} is defined by the requirement that it satisfy the two rules:

\[
(\exists x : P(x)) \equiv P(\texttt{CHOOSE } x : P(x))
\]

\[
(\forall x : P(x) = Q(x)) \Rightarrow ((\texttt{CHOOSE } x : P(x)) = \texttt{CHOOSE } x : Q(x))
\]

for any operators \( P \) and \( Q \). For a tuple of identifiers, \texttt{CHOOSE} is defined by:

\[
\texttt{CHOOSE } \langle x_1, \ldots, x_n \rangle : p \triangleq \texttt{CHOOSE } y : \exists x_1, \ldots, x_n : (y = \langle x_1, \ldots, x_n \rangle) \land p
\]

where \( y \) is an identifier different from the \( x_i \) that does not occur in \( p \).

The second of these rules allows us to deduce the equality of certain \texttt{CHOOSE} expressions that we might expect to be different. In particular, for any operator \( P \), if there exists no \( x \) satisfying \( P(x) \), then \texttt{CHOOSE } x : P(x) equals the unique value \texttt{CHOOSE } x : \texttt{FALSE}. Since the \texttt{Reals} module defines division by

\[
a/b \triangleq \texttt{CHOOSE } c \in \texttt{Real} : a = b \times c
\]

this implies that \( r/0 = s/0 \) for any nonzero real numbers \( r \) and \( s \). We can avoid this by defining an operator \texttt{Choice} so that \texttt{Choice}(\( P \), \( v \)) equals \texttt{CHOOSE } x : P(x) if there exists an \( x \) satisfying \( P(x) \), and otherwise equals some arbitrary value that depends on \( v \). One possible definition of \texttt{Choice} is

\[
\texttt{Choice}(P(c), v) \triangleq \text{if } \exists x : P(x) \text{ then } \texttt{CHOOSE } x : P(x) \text{ else } (\texttt{CHOOSE } x : x.a = v).b
\]

We could then define division by

\[
a/b \triangleq \text{let } P(c) \triangleq a = b \times c \text{ in } \texttt{Choice}(P, a)
\]

With this definition, we cannot deduce any relation between \( 1/0 \) and \( 2/0 \).

\section*{15.1.2 The Three Interpretations of Boolean Operators}

The meaning of a Boolean operator when applied to Boolean values is a standard part of traditional mathematics. I assume you know that \texttt{TRUE \& FALSE} equals \texttt{FALSE}. However, because TLA\textsuperscript+ is untyped, an expression like \( 2 \land \langle 5 \rangle \) is legal. We must therefore specify what it means. There are three ways of doing this, which I call the \texttt{conservative}, \texttt{moderate}, and \texttt{liberal} interpretations.

In the conservative interpretation, the value of an expression like \( 2 \land \langle 5 \rangle \) is completely unspecified. It could equal \( \sqrt{2} \). It need not equal \( \langle 5 \rangle \land 2 \). Hence, the ordinary laws of logic, such as the commutativity of \( \land \), are valid only for Boolean values.

In the liberal interpretation, the value of \( 2 \land \langle 5 \rangle \) is specified to be a Boolean. It is not specified whether it equals \texttt{TRUE} or \texttt{FALSE}. However, all the ordinary
laws of logic, such as the commutativity of $\land$, are valid. Hence, $2 \land \langle 5 \rangle$ equals $\langle 5 \rangle \land 2$. More precisely, any tautology of propositional or predicate logic, such as

$$(\forall x : p) \equiv \lnot(\exists x : \lnot p)$$

is valid, even if $p$ is not necessarily a Boolean for all values of $x$.\(^1\) It is easy to show that the liberal approach is sound. For example, one way of constructing operators that satisfy the liberal interpretation is to consider any nonBoolean value to be equivalent to $\text{FALSE}$.

The moderate interpretation lies between the conservative and liberal interpretations. It assumes only that expressions involving $\text{FALSE}$ and $\text{TRUE}$ have their expected values—for example, $\text{FALSE} \land 2$ equals $\text{FALSE}$, $\text{FALSE} \Rightarrow \text{"a"}$ equals $\text{TRUE}$, and $\forall n \in \text{Nat} : f[n]$ equals $\text{FALSE}$ if $f[7]$ equals $\text{FALSE}$. However, the value of $\langle 5 \rangle \land 2$ is completely unspecified. The moderate interpretation makes it easier to use Boolean-valued functions or records. For example, suppose $f$ is a Boolean-valued function. In the moderate interpretation, the expression

$$(x \in \text{DOMAIN } f) \land f[x]$$

equals $\text{FALSE}$ if $x$ is not in the domain of $f$. With the conservative interpretation, its value is unspecified in that case. If we want the expression to equal $\text{FALSE}$ when $x \notin \text{DOMAIN } f$, we would have to write it as

$$(x \in \text{DOMAIN } f) \land (f[x] = \text{TRUE})$$

The conservative interpretation is philosophically more satisfying, since it makes no assumptions about a silly expression like $2 \land \langle 5 \rangle$. However, it has the disadvantage that, when writing proofs, one has to check that all the relevant values are Booleans before applying any of the ordinary rules of logic. This can be burdensome in practice.

The semantics of TLA\(^+\) assert that the rules of the conservative interpretation are valid. The liberal interpretation is neither required nor forbidden. You should write specifications that make sense under the conservative interpretation or, if you are using Boolean-valued functions, under the moderate interpretation. However, you (and the implementer of a tool) are free to use the liberal interpretation if you wish.

### 15.1.3 Miscellaneous Constructs

The if/then/else construct was introduced on page 16 of Section 2.2. Its general form is:

```plaintext
IF p THEN e₁ ELSE e₂
```

\(^1\)Equality ($\equiv$) is not an operator of propositional or predicate logic; this tautology need not be valid for nonBoolean values if $\equiv$ is replaced by $=$.
It is defined to equal:

\[
\text{CHOOSE } v : (p \Rightarrow v = e_1) \land (\neg p \Rightarrow v = e_2)
\]

where \( v \) is an identifier that does not occur in \( p, e_1, \) or \( e_2 \).

\( \text{TLA}^+ \) provides a \textsc{case} construct with one of the following two forms:

\[
\text{CASE } p_1 \rightarrow e_1 \square \ldots \square p_n \rightarrow e_n
\]

\[
\text{CASE } p_1 \rightarrow e_1 \square \ldots \square p_n \rightarrow e_n \square \text{OTHER } \rightarrow e
\]

If some \( p_i \) is true, then these expressions equal \( e_j \) for some \( j \) such that \( p_j \) is true. (If \( p_j \) is true for more than one value of \( j \), it is not specified which of these values is chosen.) If no \( p_i \) is true, then the value is unspecified in the first form, and equals \( e \) in the second form. The second form is defined to equal:

\[
\text{CASE } p_1 \rightarrow e_1 \square \ldots \square p_n \rightarrow e_n \square \neg(p_1 \lor \ldots \lor p_n) \rightarrow e
\]

The first form is defined to equal:

\[
\text{CHOOSE } v : (p_1 \land (v = e_1)) \lor \ldots \lor (p_n \land (v = e_n))
\]

where \( v \) is an identifier that does not occur in any \( p_i \) or \( e_i \).

The \textsc{let/in} construct was introduced on page 59 of Section 5.6. Its precise meaning is explained in Section 16.4. Its general form is:

\[
\text{LET } d_1 \triangleq e_1 \ldots d_n \triangleq e_n \text{ IN } e
\]

where each \( d_i \) is either an identifier or has the form \( \text{id}(id_1, \ldots, id_k) \) for identifiers \( id, id_1, \ldots, id_k \). This expression is equivalent to

\[
\text{LET } d_1 \triangleq e_1 \text{ IN } (\text{LET } \ldots d_n \triangleq e_n \text{ IN } e)\ldots)
\]

The expression \textsc{let } \( d \triangleq f \text{ IN } e \) equals \( e \) in the context of the definition \( d \triangleq f \).

### 15.1.4 The Operators of Set Theory

The operators of set theory were introduced in Sections 1.2 and 6.1. \( \text{TLA}^+ \) is based on Zermelo-Fr"ankel set theory, in which every value is a set. Using the Boolean operators of Section 15.1.1, all the operators of set theory can be defined in terms of the single primitive operator \( \in \). For example set union can be defined by

\[
S \cup T \triangleq \text{CHOOSE } U : \forall x : (x \in U) \equiv (x \in S) \lor (x \in T)
\]

Instead of writing the definition like this, we say that \( S \cup T \) is defined by:

\[
\forall x : (x \in (S \cup T)) \equiv (x \in S) \lor (x \in T)
\]

\( \text{TLA}^+ \) provides the following operators on sets:
15.1. CONSTANT OPERATORS

\( e_1 = e_2 \) equals \( \forall x : (x \in S) \equiv (x \in T) \). (Two sets are equal if and only if they have the same elements.)

\( e_1 \neq e_2 \) equals \( \neg (e_1 = e_2) \).

\( e \in S \) A primitive, undefined expression. Intuitively, it means that \( e \) is an element of \( S \).

\( e \notin S \) equals \( \neg (e \in S) \).

\( S \cup T \) is defined by \( \forall x : (x \in (S \cup T)) \equiv (x \in S) \lor (x \in T) \).

\( S \cap T \) is defined by \( \forall x : (x \in (S \cap T)) \equiv (x \in S) \land (x \in T) \).

\( S \subseteq T \) equals \( \forall x : (x \in S) \Rightarrow (x \in T) \).

\( S \setminus T \) is defined by \( \forall x : (x \in (S \setminus T)) \equiv (x \in S) \land (x \notin T) \).

\( \{e_1, \ldots, e_n\} \) equals \( \{e_1\} \cup \ldots \cup \{e_n\} \), where \( \{e\} \) is defined by:

\[ \forall x : (x \in \{e\}) \equiv (x = e) \]

\( \{x \in S : p\} \)

where \( x \) is a bound identifier or a tuple of bound identifiers. The expression \( S \) is outside the scope of the bound identifier(s). For \( x \) an identifier, the expression is defined by

\[ \forall y : (y \in \{x \in S : p\}) \equiv (y \in S) \land \hat{p} \]

where the identifier \( y \) does not occur in \( S \) or \( p \), and \( \hat{p} \) is \( p \) with \( y \) substituted for \( x \). For \( x \) a tuple, the expression is defined by

\[ \{\langle x_1, \ldots, x_n \rangle \in S : p\} = \]

\[ \{y \in S : (\exists x_1, \ldots, x_n : (y = \langle x_1, \ldots, x_n \rangle) \land p)\} \]

where \( y \) is an identifier different from the \( x_i \) that does not occur in \( S \) or \( p \). See Section 15.1.7 for further details about tuples.

\( \{e : y_1 \in S_1, \ldots, y_n \in S_n\} \)

where each \( y_i \) has the form \( x_1, \ldots, x_k \) or \( \langle x_1, \ldots, x_k \rangle \), and each \( x_j \) is an identifier that is bound in the expression. The expressions \( S_i \) lie outside the scope of the bound identifiers. The simple form \( \{e : x \in S\} \), for \( x \) an identifier, is defined by

\[ \forall y : (y \in \{e : x \in S\}) \equiv (\exists x \in S : e = y) \]
The general form is reduced to this simple form by the following relations:

\[
\{ e : y_1 \in S_1, \ldots, y_n \in S_n \} =
\{ e : x_1 \in S, \ldots, x_n \in S \}
\]

\[
\{ e : \langle x_1, \ldots, x_n \rangle \in S \} =
\{ (\text{LET } z \triangleq \text{choose } \langle x_1, \ldots, x_n \rangle : y = \langle x_1, \ldots, x_n \rangle)
\]

\[
x_1 \triangleq z[1]
\]

\[
\vdots
\]

\[
x_n \triangleq z[n] \text{ IN } e : y \in S
\]

where the \( x_i \) are identifiers, and \( y \) and \( z \) are identifiers distinct from the \( x_i \) that do not occur in \( e \) or \( S \). See section 15.1.7 for further details about tuples.

**SUBSET** \( S \) is defined by \( \forall T : (T \in \text{SUBSET } S) \equiv (T \subseteq S) \)

**UNION** \( S \) is defined by \( \forall x : (x \in \text{UNION } S) \equiv (\exists T \in S : x \in T) \).

### 15.1.5 Functions

Functions were described in Section 5.2, and recursive function definitions were introduced in Section 5.5 and discussed in Section 6.3. Mathematicians traditionally define a function to be a set of pairs. TLA\(^+\) defines pairs (and all tuples) to be functions, and takes function to be a primitive concept. We can define the operator \( \text{IsAFcn} \) as follows such that \( \text{IsAFcn}(f) \) is true iff \( f \) is a function:

\[
\text{IsAFcn}(f) = f = [x \in \text{DOMAIN } f \mapsto f[x]]
\]

Two functions are equal iff they have the same domain and they assign the same value to each element in their domain:

\[
\forall f, g : \text{IsAFcn}(f) \land \text{IsAFcn}(g) \Rightarrow ((f = g) \equiv \land \text{DOMAIN } f = \text{DOMAIN } g
\land \forall x \in \text{DOMAIN } f : f[x] = g[x])
\]

TLA\(^+\) defines a function of multiple arguments to be a function whose domain is a Cartesian product, where \( f[e_1, \ldots, e_n] \) is just an abbreviation for \( f([e_1, \ldots, e_n]) \).

The TLA\(^+\) operators for manipulating functions are:

\[
f[e_1, \ldots, e_n]
\]

where the \( e_i \) are expressions. For \( n = 1 \), this is a primitive, undefined expression. Intuitively, it is the result of applying the function \( f \) to the
expression $e_1$. It is unspecified if $f$ is not a function or $e_1$ is not in its domain. For $n > 1$, the expression is defined by

$$f[e_1, \ldots, e_n] = f[\langle e_1, \ldots, e_n \rangle]$$

See Section 15.1.7 for the definition of the tuple $\langle e_1, \ldots, e_n \rangle$.

**DOMAIN**

is a primitive, undefined expression. It is the domain of $f$, if $f$ is a function.

$$\left[ y_1 \in S_1, \ldots, y_n \in S_n \mapsto e \right]$$

where each $y_i$ has the form $x_1, \ldots, x_k$ or $\langle x_1, \ldots, x_k \rangle$, and each $x_j$ is an identifier that is bound in the expression. The expressions $S_i$ lie outside the scope of the bound identifiers. The simple form $\left[ x \in S \mapsto e \right]$, for $x$ an identifier, is effectively defined by the two axioms:

$$\left( \text{DOMAIN} \left[ x \in S \mapsto e \right] \right) = S$$

$$\forall y \in S : \left[ x \in S \mapsto e \right][y] = \hat{e}$$

where $y$ is an identifier different from $x$ that does not occur in $S$ or $e$, and $\hat{e}$ is the expression obtained from $e$ by substituting $y$ for $x$. The general form of the expression can be reduced to the simple form by using the following relations:

$$\left[ x_1 \in S_1, \ldots, x_n \in S_n \mapsto e \right] =$$

$$\left[ \langle x_1, \ldots, x_n \rangle \in S_1 \times \ldots \times S_n \mapsto e \right]$$

$$\left[ \ldots y_{i-1} \in S_{i-1}, \ , x_1, \ldots, x_k \in S_i, \ y_{i+1} \in S_{i+1} \ldots \mapsto e \right] =$$

$$\left[ \ldots y_{i-1} \in S_{i-1}, \ , x_1 \in S_i, \ldots, x_k \in S_i, \ y_{i+1} \in S_{i+1} \ldots \mapsto e \right]$$

$$\left[ \ldots y_{i-1} \in S_{i-1}, \ , \langle x_1, \ldots, x_k \rangle \in S_i, \ y_{i+1} \in S_{i+1} \ldots \mapsto e \right] =$$

$$\left[ \ldots y_{i-1} \in S_{i-1}, \ , y \in S_i, \ y_{i+1} \in S_{i+1} \ldots \mapsto \text{LET } z, \ \hat{z} \equiv \text{CHOOSE} \langle x_1, \ldots, x_n \rangle : y = \langle x_1, \ldots, x_n \rangle \right.$$  

$$x_1 \triangleq z[1]$$

$$\vdots$$

$$x_n \triangleq z[n] \ \text{IN } e]$$

where $y$ and $z$ are identifiers distinct from the $x_i$ that do not occur in $e$ or in any of the $y_j$ or $S_j$. See Section 15.1.7 for details about tuples.

$\left[ S \rightarrow T \right]$ is defined by

$$\forall f : f \in \left[ S \rightarrow T \right] \equiv$$

$$\text{IsAFcn}(f) \land (S = \text{domain} f) \land (\forall x \in S : f[x] \in T)$$

where $x$ and $f$ do not occur in $S$ or $T$, and $\text{IsAFcn}$ is defined above.

$$\left[ f \text{ except } \langle a_1 = e_1, \ldots, a_n = e_n \rangle \right]$$

where each $a_i$ has the form $[d_1] \ldots [d_k]$ and each $d_j$ is an expression. The
special operator name $\oplus$ may occur in the $e_i$, but not in the $a_i$. For the simple case when $n = 1$ and $a_1$ is $[d]$, this is defined by

\[
[f \text{ except } !d = e] =
\begin{align*}
[y \in \text{domain } f \rightarrow \text{IF } y = d \text{ THEN LET } \oplus \triangleq f[d] \text{ IN } e \\
\text{ELSE } f[y]
\end{align*}
\]

where $y$ does not occur in $f$, $d$, or $e$. The general form is reduced to this simple case by using the relations:

\[
[f \text{ except } !a_1 = e_1, \ldots, !a_n = e_n] =
\end{align*}
\[
[[f \text{ except } !a_1 = e_1, \ldots, !a_{n-1} = e_{n-1}] \text{ EXCEPT } !a_n = e_n]
\]

\[
[f \text{ except } ![d_1] \ldots ![d_k] = e] =
\end{align*}
\[
[f \text{ except } ![d_1] = ![\oplus \text{ EXCEPT } ![d_2] \ldots ![d_k] = e]]
\]

TLA$^+$ also allows function definitions of the form

\[
f[y_1 \in S_1, \ldots, y_n \in S_n] \triangleq e
\]

This definition is equivalent to

\[
f \triangleq \text{choose } f : f = [y_1 \in S_1, \ldots, y_n \in S_n \rightarrow e]
\]

Note that the identifier $f$ may occur in the expression $e$.

### 15.1.6 Records

Records were introduced in Section 3.2 and further explained in Section 5.2. A record is a function whose domain is a finite set of strings, where $r.h$ means $r[\text{"h"}]$, for a record component $h$. A record component is syntactically an identifier. In the ASCII version, it is a string of letters, digits, and the underscore character (_) that contains at least one letter. The record constructs are defined as follows in terms of the function constructs.

\[
e.h \text{ is defined to equal } e[\text{"h"}].
\]

\[
[h_1 \rightarrow e_1, \ldots, h_n \rightarrow e_n] \text{ is defined to equal}
\]

\[
[y \in \{\text{"h}_1\}, \ldots, \text{"h}_n\} \rightarrow 
\text{CASE } (y = \text{"h}_1) \rightarrow e_1 \square \ldots \square (y = \text{"h}_n) \rightarrow e_n
\]

where $y$ does not occur in any of the expressions $e_i$.

\[
[h_1 : S_1, \ldots, h_n : S_n] \text{ is defined to equal}
\]

\[
\{[h_1 \rightarrow y_1, \ldots, h_n \rightarrow y_n] : y_1 \in S_1, \ldots, y_n \in S_n\}
\]

where the $y_i$ do not occur in any of the expressions $S_j$. 

[r \text{ EXCEPT } !a_1 = e_1, \ldots, !a_n = e_n]
where $a_i$ has the form $b_1 \ldots b_k$ and each $b_j$ is either (i) $[d]$, where $d$ is an expression, or (ii) $h$, where $h$ is a record component. It is defined to equal the corresponding function \text{EXCEPT} construct in which each $h$ is replaced by “[h”].

### 15.1.7 Tuples

Tuples are explained in Section 5.4. As stated there, an $n$-tuple is a function whose domain is the set $1 \ldots n$, which equals $\{j \in \text{Nat} : (0 < j) \land (j \leq n)\}$. Thus, the TLA$^+$ expression $(a, b, c)[2]$ equals $b$. (Numbers are discussed in Section 15.1.9 below.) Here are the meanings of operators on tuples:

- $e[i]$ is explained in Section 15.1.5.
- $\langle e_1, \ldots, e_n \rangle$ is defined to equal
  
  \[\{i \in 1 \ldots n \mapsto e_i\}\]

  where $i$ does not occur in any of the expressions $e_j$.

- $S_1 \times \cdots \times S_n$ is defined to equal
  
  \[\{\langle y_1, \ldots, y_n \rangle : y_1 \in S_1, \ldots, y_n \in S_n\}\]

  where the identifiers $y_i$ do not occur in any of the expressions $S_j$.

### 15.1.8 Strings

TLA$^+$ defines a string to be a tuple of characters. Thus, “abc” equals

\[\langle \text{“abc”}[1], \text{“abc”}[2], \text{“abc”}[3] \rangle\]

TLA$^+$ does not specify what a character is. However, it does specify that different characters (those having different computer representations) are different. Thus “a”[1], “b”[1], and “A”[1] (the characters $a$, $b$, and $A$) are all different. The built-in operator \text{STRING} is defined to be the set of all strings.

Although TLA$^+$ doesn’t specify what a character is, it’s easy to define operators that assign values to characters. For example, here’s the definition of an operator \text{ASCII} that assigns to every lower-case letter its ASCII representation.\footnote{This clever way of using \text{CHOOSE} to map from characters to numbers was pointed out to me by Georges Gonthier.}

\[\text{ASCII}(\text{char}) \triangleq 96 + \text{CHOOSE } i \in 1 \ldots 26 : \text{“abcdefghijklmnopqrstuvwxyz”}[i] = \text{char}\]
This defines $ASCII(\text{"a"}[1])$ to equal 97, the ASCII code for the letter $a$, and $ASCII(\text{"z"}[1])$ to equal 122, the ASCII code for $z$. Section 14.1.2) on page 168) illustrates how a specification can make use of the fact that strings are tuples.

Exactly what characters may appear in a string is system-dependent. A Japanese version of TLA$^+$ might not allow the character $a$. The standard ASCII version contains the following characters:

```
abcdefghijklmnopqrstuvwxyz
ABCDEFGHIJKLMNOPQRSTUVWXYZ
0123456789
~ @ # $ % ^ & * - + = ( ) [ ] < > / \ , . ? ; : " '
```

plus the space character. Since strings are delimited by a double-quote ('"'), some convention is needed for typing a string that contains a double-quote. Moreover, conventions are also needed to type characters like \texttt{\textbackslash lf}. In the ASCII version of TLA$^+$, the following pairs of characters, beginning with a \textbackslash character, are interpreted in a string as follows:

```
\" " \t \langle HT\rangle \\l " \l " \langle LF\rangle \langle FF\rangle \langle CR\rangle
```

With this convention, "a\"\l "b\"" represents the string consisting of the five-character sequence $a \text{"} b$.

### 15.1.9 Numbers

TLA$^+$ defines a sequence of digits like 63 to be the usual natural number represented in decimal notation. More precisely, it defined to be the element \texttt{Succ(Succ(\ldots(Zero)\ldots))} in the set \texttt{Nat}, defined in the standard \texttt{Peano} module described in Section 17.5 below. TLA$^+$ also allows the binary representation \texttt{\textbackslash b111111}, the octal representation \texttt{\textbackslash o77}, and the hexadecimal representation \texttt{\textbackslash h3F} of that number. (Case is ignored in the prefixes and in the hexadecimal representation, so \texttt{\textbackslash H3F} and \texttt{\textbackslash h3f} are equivalent to \texttt{\textbackslash h3F}.)

A decimal number like 3.14159 is defined to be the usual rational number $314159/10^5$. More precisely, it is that element of the set \texttt{Real} of real numbers defined by the standard \texttt{Reals} modules described in Section 17.5.

Numbers are built into TLA$^+$, so 63 is defined even in a module that does not extend or instantiate one of the standard numbers modules. However, + is not built in. You can write a module that defines + any way you want, in which case $40 + 23$ need not equal 63. Of course, $40 + 23$ does equal 63 for + defined by the standard numbers modules.
15.2. Nonconstant Operators

The nonconstant operators are what distinguish TLA\(^+\) from “ordinary mathematics”. There are two classes of nonconstant operators: action operators, listed in Figure 15.3 on this page, and temporal operators, listed in Figure 15.4 on this page. Before defining their meanings, we must first understand the semantics of state functions.

15.2.1 The Meaning of a State Function

A constant expression is one containing only constant operators and declared constants. Section 15.1 defines the meaning of the constant operators of TLA\(^+\), which defines the meaning of a constant expression in terms of the meanings of the declared constants. The meaning of a declared constant is an undefined primitive. We write the meaning of a declared constant \(c\) as \([c]\).

The semantics of nonconstant operators is defined in terms of the concept of state. As explained in Sections 2.1 and 2.3, a state is an assignment of values to variables. Let \(s[x]\) be the value that state \(s\) assigns to variable \(x\). For a constant expression \(e\), we define \(s[e]\) to equal \([e]\). This expresses formally that a constant has the same value in all states.

State functions were defined on page 25 of Section 3.1. A state function is an expression that contains variables and constant operators. Semantically, a state function \(e\) assigns a value \(s[e]\) to a state \(s\). (We thus say both that state

\[\Box F\] \(F\) is always true\)
\[\Diamond F\] \(F\) is eventually true\)
\(WF_e(A)\) Weak fairness for action \(A\)
\(SF_e(A)\) Strong fairness for action \(A\)
\(F \leadsto G\) \(F\) leads to \(G\)
\(F \Rightarrow G\) \(F\) guarantees \(G\)
\(\exists x : F\) Temporal existential quantification (hiding).]
\(\forall x : F\) Temporal universal quantification.]

Figure 15.4: The temporal operators of TLA\(^+\).
s assigns \(s[x]\) to variable \(x\), and that \(x\) assigns \(s[x]\) to state \(s\).) Since a state function is built from variables, constants, and constant operators, we can define \(s[e]\) inductively in terms of its value when \(e\) is a constant or a variable. For example, \(s[e_1 \cup e_2]\) equals \(s[e_1] \cup s[e_2]\).

15.2.2 Action Operators

A transition function is an expression built from state functions using the action operators of TLA+. Semantically, a transition function \(e\) assigns a value to any pair of states \(s; t\). (As explained in Section 2.2, a pair of states is called a step. I’ll now write this pair as \(s; t\) instead of \(s \rightarrow t\).) An action is a Boolean-valued transition function. When I describe \(A\) as an action—for example, in the definition of \(\text{Enabled } A\)—it means that \(A\) is a transition function, but the expression is silly if \(A\) is not an action.

Priming and the construct \([A]_e\) were introduced in Section 2.2, and the action-composition operator “\(\cdot\)” was introduced in Section 7.3. The other action operators were introduced in Chapter 8. The definitions of all the action operators are:

\[
e' \quad \text{is defined by } \langle s, t \rangle[e'] \triangleq t[e] \text{ for a state function } e.
\]

\[
[A]_e \quad \text{equals } A \lor (e' = e).
\]

\[
\langle A \rangle_e \quad \text{equals } A \land (e' \neq e).
\]

\[
\text{Enabled } A \quad \text{is the state function defined by}
\]

\[
s[\text{Enabled } A] \triangleq \exists t : \langle s, t \rangle[A]
\]

for an action \(A\). (The quantification is over all states \(t\).)

\[
\text{UNCHANGED } e \quad \text{equals } e' = e, \text{ for a state function } e.
\]

\[
A \cdot B \quad \text{is the action defined by}
\]

\[
\langle s, t \rangle[A \cdot B] \triangleq \exists u : \langle s, u \rangle[A] \land \langle u, t \rangle[B]
\]

for actions \(A\) and \(B\). (The quantification is over all states \(u\).)

15.2.3 Temporal Operators

As explained in Section 8.1, a temporal formula \(F\) is true or false on a behavior, where a behavior is a sequence of states. The truth value of \(F\) on a behavior \(\sigma\) is written \(\sigma \models F\).

A temporal formula is built from transition functions using the temporal operators of TLA+. The temporal operators of TLA+ are explained in Chapter 8—except for \(\neg p\), which is explained in Chapter 10.2. We define the meaning
of a temporal formula in terms of the meaning of an action. If $A$ is an action, we define $\sigma \models A$ to equal \((s_0, s_1)[A]\), where $s_0$ and $s_1$ are the first two states of the behavior $\sigma$.

$\Box F$ is defined by $\sigma \models F \iff \forall n \in \text{Nat} : (\sigma^+ \models F)$, where $\sigma^n$ is the suffix of $\sigma$ obtained by deleting its first $n$ states.

$\Diamond F$ is defined to equal $\neg \Box \neg F$.

$WF_e(A)$ is defined to equal $\Box \Diamond \neg (\text{ENABLED } A) \lor \Box \Diamond (A)_v$

$SF_e(A)$ is defined to equal $\Diamond \Box \neg (\text{ENABLED } A) \lor \Diamond \Box (A)_v$

$F \rightsquigarrow G$ is defined to equal $\Box (F \Rightarrow \Diamond G)$.

$F \rightleftharpoons G$ is defined by

$$\sigma \models (F \rightleftharpoons G) \iff (\sigma \models (F \Rightarrow G)) \land (\forall n : \sigma \models_n (F \Rightarrow G))$$

where $\sigma \models_n H$ is true iff there exists a behavior $\tau$ such that the first $n$ states of $\tau$ and $\sigma$ are the same, and $\tau \models H$.

$\exists x : F$ is defined by

$$\sigma \models (\exists x : F) \iff \exists \tau : (\tau \sim_x \sigma) \land \sigma \models F$$

where $\tau \sim_x \sigma$ means that $\tau$ and $\sigma$ are equivalent except for stuttering steps and the values assigned by their states to the variable $x$. (See Section 8.6.)

$\forall x : F$ is defined to equal $\neg \exists x : \neg F$. 
Chapter 16

The Meaning of a Module

A TLA\(^+\) specification is a collection of modules. This chapter explains the higher-level structure of a module—the structure above the level of an expression. It describes the meaning of a module, assuming that the meaning of an expression containing only the built-in operators of TLA\(^+\) is already known. The meaning of these operators is described in Chapter 15. The chapter also defines the “semantic” part of the syntax of TLA\(^+\), the syntactic conditions not described in Chapter 14.

We consider meaning independent of the particular syntax with which it is expressed. Examples use the actual TLA\(^+\) syntax and assume that you understand the approximate meanings of the constructs that are explained in Part I.

16.1 Operators and Expressions

16.1.1 Orders of Operators

TLA\(^+\) allows us to define three classes of operators—0th-, 1st-, and 2nd-order operators.

- \( A \triangleq x + y \) defines \( A \) to be the 0th-order operator \( x + y \). A 0th-order operator takes no arguments, so it is an ordinary expression. We say that it has arity \( \_ \).
- \( F(x, y) \triangleq x \cup \{z, y\} \) defines \( F \) to be a 1st-order operator. For any expressions \( e_1 \) and \( e_2 \), it defines \( F(e_1, e_2) \) to be an expression. We say that \( F \) has arity \( \langle \_, \_ \rangle \).

In general, a 1st-order operator takes expressions (0th-order operators) as arguments. Its arity is the tuple \( \langle \_, \ldots, \_ \rangle \), with one \( \_ \) for each argument.
• \( G(f(\_ , \_), x, y) \triangleq f(x, \{x, y\}) \) defines \( G \) to be a 2nd-order operator. The operator \( G \) takes three arguments—its first argument is a 1st-order operator that takes two arguments; its last two arguments are expressions (0th-order operators). For any operator \( op \) of arity \( \langle \_ , \_ \rangle \), and any expressions \( e_1 \) and \( e_2 \), this defines \( G(op, e_1, e_2) \) to be an expression. We say that \( G \) has arity \( \langle \_ , \_ \rangle \).\(^1\)

In general, the arguments of a 2nd-order operator may be 0th- or 1st-order operators. A 2nd-order operator has an arity of the form \( \langle a_1, \ldots, a_n \rangle \), where each \( a_i \) is either \( \_ \) or \( \langle \_ , \_ \rangle \).

\( TLA^+ \) does not permit 3rd-order or higher-order operators.\(^2\)

### 16.1.2 \( \lambda \) Expressions

When we define a 0th-order operator \( A \) by \( A \triangleq exp \), we can write what the operator \( A \) equals—it equals the expression \( exp \). However, when we define the 1st-order operator \( F \) by

\[
F(x, y) \triangleq x \cup \{z, y\}
\]

we can’t write what \( F \) equals. We now extend expressions to \( \lambda \) expressions, and we write the operator that \( F \) equals as:

\[
\lambda x, y : x \cup \{z, y\}
\]

The symbols \( x \) and \( y \) in this \( \lambda \) expression are called \( \lambda \) parameters. We use \( \lambda \) expressions only to help explain the meaning of \( TLA^+ \) specifications; we can’t write a \( \lambda \) expression in \( TLA^+ \).

We also allow 2nd-order \( \lambda \) expressions, so the operator \( G \) defined by

\[
G(f(\_ , \_), x, y) \triangleq f(y, \{x, y\})
\]

is equal to the \( \lambda \) expression

(16.1) \( \lambda f(\_ , \_), x, y : f(y, \{x, y\}) \)

A \( \lambda \) parameter is a bound identifier, just like the identifier \( x \) in \( \forall x : F \). As with any bound identifiers, renaming the \( \lambda \) parameters in a \( \lambda \) expression doesn’t change the meaning of the expression. For example, (16.1) is equivalent to

\[
\lambda abc(\_ , \_), qq, m : abc(m, \{qq, m\})
\]

\(^1\)Even though it allows 2nd-order operators, \( TLA^+ \) is still what logicians call a first-order logic because it permits quantification only over 0th-order operators. We can’t write the second-order formula \( \exists x(\_ ) : exp \) in \( TLA^+ \).

\(^2\)In principle, there’s no reason not to allow such higher-order operators. However, they would complicate the level checking described in Section 16.2.
Logicians call this kind of renaming \(\alpha\) conversion.

If \(Op\) is the \(\lambda\) expression \(\lambda p_1, \ldots, p_n : exp\), then \(Op(e_1, \ldots, e_n)\) equals the result of replacing each \(\lambda\) parameter \(p_i\) in \(exp\) with \(e_i\). For example,

\[
(\lambda x, y : x \cup \{z, y\})(TT, w + z) = TT \cup \{z, (w + z)\}
\]

This procedure for “evaluating” a \(\lambda\) expression application is called \(\beta\) reduction by logicians.

For uniformity, we consider an expression \(exp\) to be the same as a \(\lambda\) expression \(\lambda : exp\) with no parameters. We consider this to be the \(n = 0\) case of \(\lambda p_1, \ldots, p_n : exp\).

### 16.1.3 Simplifying Operator Application

For simplicity, I will assume that every operator application has the form \(Op(e_1, \ldots, e_n)\). TLA\(^+\) provides a number of different syntaxes for operator application, so I need to explain how they are translated into this form. These other syntaxes fall into the following classes:

- **Simple constructs with a fixed number of arguments.** These include infix operators like \(+\), and constructs like \(\text{If/Then/Else}\) and function application. Such constructs pose no problem. We can write \(+a, b\) instead of \(a + b\), \(\text{IfThenElse}(p, e_1, e_2)\) instead of \(\text{If } p \text{ Then } e_1 \text{ Else } e_2\) and \(\text{Apply}(f, e)\) instead of \(f[e]\). An expression like \(a + b + c\) is an abbreviation for \((a + b) + c\), so it can be written \((+(a, b), c)\).

- **Simple constructs with a variable number of arguments.** These include constructs like \(\{e_1, \ldots, e_n\}\) and \([h_1 \mapsto e_1, \ldots, h_n \mapsto e_n]\). We can consider each of these constructs to be repeated application of simpler operators with a fixed number of arguments. For example,

\[
\{e_1, e_2, e_3\} = (\{e_1\} \cup \{e_2\}) \cup \{e_3\}
\]

and \(\{e\}\) can be written \(\text{Singleton}(e)\).

- **Constructs that introduce bound variables**—for example,

\[
\exists x \in S : x + z > y
\]

We can rewrite this expression as

\[
\text{ExistsIn}(S, \lambda x : x + z > y)
\]

where \(\text{ExistsIn}\) is a 2nd-order operator of arity \(\{\_\}, \{\_\}\). As explained in Section 15.1.1, all the variants of the \(\exists\) construct can be expressed using either \(\exists x \in S : e\) or \(\exists x : S\). All the other constructs that introduce bound variables, such as \(\{x \in S : \text{exp}\}\) can similarly be expressed in the form
$Op(e_1, \ldots, e_n)$ using $\lambda$ expressions and 2nd-order operators $Op$. (Constructs like $\exists (x, y) \in S : exp$ that involve a tuple of bound identifiers are expressed in terms of ordinary bound identifiers as described in Chapter 15.)

- Operator applications such as $M(x)! Op(y, z)$ that arise from instantiation. We write this as $M! Op(x, y, z)$.
- $\text{LET}$ expressions. We will describe the meaning of a $\text{LET}$ expression in Section 16.4. For now, we don’t consider them.

## 16.1.4 Expressions

We can now inductively define an expression to be either a 0th-order operator, or to have the form $Op(e_1, \ldots, e_n)$ where $Op$ and each $e_i$ is either an expression or a 1st-order operator. If $Op$ has arity $(a_1, \ldots, a_n)$, then each $e_i$ must have arity $a_i$. This means that $e_i$ is an expression if $a_i$ equals 0, otherwise it is a 1st-order operator with arity $a_i$. We require that $Op$ not be a $\lambda$ expression. (If it is, we can use $\beta$ reduction to “evaluate” $Op(e_1, \ldots, e_n)$ and eliminate the $\lambda$ expression $Op$.) Hence, a $\lambda$ expression can appear in an expression only as an argument of a 2nd-order operator. This implies that only 1st-order $\lambda$ expressions can appear in an expression.

We have eliminated all bound identifiers except the ones in $\lambda$ expressions. We maintain the TLA$^+$ requirement that an identifier that already has a meaning cannot be used as a bound identifier. Thus, in $\lambda p_1, \ldots, p_n : exp$, no $\lambda$ expression that is a subexpression of $exp$ may use any of the $p_i$ as a $\lambda$ parameter.

Remember that $\lambda$ expressions are used only to explain the semantics of TLA$^+$. They are not part of the language, and they can’t be used in a TLA$^+$ specification.

## 16.2 Levels

In TLA, an expression has one of four basic levels, represented by the natural numbers 0, 1, 2, and 3. These levels are described below, using examples that assume $x$, $y$, and $c$ are declared by

```
VARIABLES x, y
CONSTANT c
```

and symbols like $+$ have their usual meanings.

0. A $\text{constant}$-level expression contains only constants and constant operators. Example: $c + 3$.

1. A $\text{variable}$-level expression may contain constants and constant operators and unprimed variables. Example: $x + 2 * c$. 

2. An action-level expression may contain anything except temporal operators. Example: \( x' + y > c \).

3. A temporal-level expression may contain any TLA operator. Example: \( \Box [x' > y + c]_{(x,y)} \).

Each level is included in higher levels. For example, any variable-level expression is also an action-level expression. (However, the syntax of TLA makes it impossible to use an arbitrary action-level formula as a temporal-level expression.)

TLA\(^+\) has rules that determine whether an expression is level-correct. The primary purpose of those rules is to disallow "double-primes"—that is, expressions of the form \( x'' \), which are meaningless in TLA. Those rules are expressed by defining the level of an operator. A 0th-order operator is an expression; its level is one of the four basic levels. The level of a 1st- or 2nd-order operator \( Op \) is a more complex object. It consists of two things:

- For each argument of \( Op \), a set of allowed levels. The expression \( Op(e_1, \ldots, e_n) \) is level-correct if the level of each \( e_i \) is in the set of allowed levels for the \( i \)th argument.
- A rule that determines the level of \( Op(e_1, \ldots, e_n) \), given the level of each \( e_i \).

For example, if the 1st-order operator \( F \) is defined by

\[
F(x, p) \stackrel{\triangleq}{=} p \land \text{Enabled}(x' = 7)
\]

then its level asserts that \( F(e_1, e_2) \) is level-correct iff \( e_1 \) has level 0 or 1 (constant- or variable-level), and that its level is the maximum of 1 and the level of \( e_2 \).\(^3\)

I won't bother to define precisely what the level of an operator is and what the precise rules are for level-correctness. I will just describe the important class of constant operators. This class includes most of the built-in operators of TLA\(^+\), such as \( \cup \) and the operators that describe ordinary quantification. A constant operator allows arguments of any level.\(^4\) If \( Op \) is a constant 1st-order operator, then the level of the expression \( Op(e_1, \ldots, e_n) \) is the maximum of the levels of all the expressions \( e_i \). If \( Op \) is a constant 2nd-order operator, then the expression \( Op(e_1, \ldots, e_n) \) has constant level if all the \( e_i \) have constant level. Any operator whose definition contains only constant operators is itself a constant operator.

The precise rules for level checking, and the rules for determining the level of a defined operator, are rather subtle. However, the subtlety arises in the handling of unusual cases that never arise in practice. (Most of those cases involve definitions of nonconstant second-order operators.) In the rest of this description of the meaning of a TLA\(^+\) specification, I will ignore level checking.

---

\(^3\)The expression \( \text{Enabled} A \) is a state predicate, for any action \( A \).

\(^4\)The semantics of TLA\(^+\) allows an expression like \( P \cup Q \) (see Section 15.1.2), but we can rule out such a nonsensical expression by defining the level of \( \cup \) to allow only arguments of level at most 2.
16.3 Contexts

Section 16.1.4 stated that $Op(e_1, \ldots, e_n)$ is an expression if $Op$ is an operator of arity $(a_1, \ldots, a_n)$ and each $e_i$ is an operator of arity $a_i$. (There are also level-correctness requirements, which I am ignoring.) If we look at expressions purely syntactically, $Op$ is not an operator, but rather an operator name—something like $+$ or $\cup$ or $Foo$. To decide if $Op(e_1, \ldots, e_n)$ is a syntactically correct expression, we need to know the arity of the operator denoted by the name $Op$. This involves the concept of a context, which I now define.

First, I must define declarations and definitions. A declaration assigns an arity and level to an operator name. A definition assigns a $\lambda$ expression to an operator name. (This $\lambda$ expression may not contain any LET construct.) A module definition assigns the meaning of a module to a module name. (The meaning of a module is defined in Section 16.5 below.) A context consists of a set of declarations, definitions, and module definitions such that:

C1. An operator name is declared or defined at most once by the context. (This means that it can’t be both declared and defined.)

C2. No operator defined or declared by the context appears as a $\lambda$ parameter in any definition’s expression.

C3. Every operator name (other than a $\lambda$ parameter) that appears in a definition’s expression is declared by the context.

C4. No module name is assigned meanings by two different module definitions.

Module and operator names are considered separately. The same string may be both a module name that is defined by a module definition and an operator name that is either declared or defined by an ordinary definition.

For our examples, I ignore the levels of operators and consider only their arities. Let $Op : a$ be a declaration that assigns arity $a$ to operator name $Op$; let $Op \triangleq e$ be a definition that assigns $\lambda$ expression $e$ to operator name $Op$; and let $M \triangleq \ldots$ be a module definition that assigns some module meaning to module name $M$. Here is an example of a context:

\[(16.2) \{ \cup : (\pi \pi), \quad a : \pi, \quad b : \pi, \quad \in : (\pi \pi), \quad c \triangleq \cup(a, b), \quad foo \triangleq \lambda p, q : \in (p, \cup(q, a)), \quad \text{Naturals} \triangleq \ldots \}\]

We now define $Op(e_1, \ldots, e_n)$ to be syntactically correct in a context $C$ iff $Op$ and every operator symbol that appears in the $e_i$ is declared or defined in $C$, and the expression is correct according to the arity rules when those operator names have the arities assigned to them by $C$. For example, if $C$ is the context (16.2), then:

- $foo(\in(a, b), \ foo(b, c))$ is correct in context $C$. 
16.4. THE MEANING OF A \lambdab\ EXPRESSION

- $\text{foo}(\cup(a, c), \in)$ is incorrect in context $C$. (The second argument of $\text{foo}$ has the wrong arity.)
- $\text{foo}(\cap(a, c), b)$ is incorrect in context $C$. (The operator symbol $\cap$ is not declared or defined in $C$.)

For a $\lambda$ expression $e$ to be syntactically correct in a context $C$, we also require that no $\lambda$ parameter that appears in $e$ is a declared or defined operator of $C$.

We also allow a context to contain a special definition of the form $\text{Op} \overset{?}{=} ?$ that assigns to the name $\text{Op}$ an “illegal” value $?$ that is not a $\lambda$ expression. This definition indicates that, in the context, it is illegal to use the operator name $\text{Op}$.

16.4 The Meaning of a $\lambda$ Expression

I now define the meaning $C[e]$ of a $\lambda$-expression $e$ in a context $C$. We allow $e$ to contain LET constructs, so this defines the meaning of LET.

We can define the meaning of something only in terms of the meaning of something else. Since mathematical definitions are noncircular, we must stop somewhere and leave the meaning of some primitive elements unspecified. In the definition of $C[e]$, I leave unspecified the meanings of the symbols declared in $C$. So, I define $C[e]$ to be a $\lambda$ expression, with no LET constructs, whose operator symbols are all declared (not defined) in the context $C$.

Basically, $C[e]$ is obtained from $e$ by replacing all defined operator names with their definitions, and then applying $\beta$ reduction whenever possible. Recall that $\beta$ reduction replaces

$$(\lambda p_1, \ldots, p_n : \text{exp})(e_1, \ldots, e_n)$$

with the expression obtained by replacing each $p_i$ with $e_i$ in $\text{exp}$. The precise inductive definition of $C[e]$ is:

- If $e$ is an operator symbol, then $C[e]$ equals (i) $e$ if $e$ is declared in $C$, or (ii) the $\lambda$ expression of $e$’s definition in $C$ if $e$ is defined in $C$.
- If $e$ is $\text{Op}(e_1, \ldots, e_n)$, where $\text{Op}$ is declared in $C$, then $C[e]$ equals the expression $C[\text{Op}][C[e_1], \ldots, C[e_1]]$.
- If $e$ is $\text{Op}(e_1, \ldots, e_n)$, where $\text{Op}$ is defined in $C$ to equal the $\lambda$ expression $d$, then $C[e]$ equals the $\beta$ reduction of $d(C[e_1], \ldots, C[e_1])$, where $d$ is obtained from $d$ by $\alpha$ reduction (replacement of $\lambda$ parameters) so that no $\lambda$ parameter appears in both $d$ and some $e_i$.
- If $e$ is $\lambda p_1, \ldots, p_n : \text{exp}$, then $C[e]$ equals $\lambda p_1, \ldots, p_n : D[\text{exp}]$, where $D$ is the context obtained by adding to $C$ the declarations that, for each $i$ in $1 \ldots n$, assigns to the $i$th $\lambda$ parameter’s name the arity determined by $p_i$. 
CHAPTER 16. THE MEANING OF A MODULE

- If $e$ is let $Op \overset{d}{=} d$ in $exp$, where $d$ is a $\lambda$ expression and $exp$ an expression, then $C[e]$ equals $D[exp]$, where $D$ is the context obtained by adding to $C$ the definition that assigns $C[d]$ to $Op$.

The last condition defines the meaning of any let construct, because:

- The operator definition $Op(p_1, \ldots, p_n) \overset{d}{=} d$ in a let means:
  
  $Op \overset{d}{=} \lambda p_1, \ldots, p_n : d$

- A function definition $Op[x \in S] \overset{d}{=} d$ in a let means:
  
  $Op \overset{d}{=} \text{choose } Op : Op = [x \in S \mapsto d]$

- The expression let $Op_1 \overset{d_1}{=} d_1 \ldots Op_n \overset{d_n}{=} d_n$ in $exp$ is defined to equal $n$ nested let constructs, each with a single definition.

From this definition of the meaning of a let expression, it should be clear what the requirements for syntactic correctness are.

Observe that if a $\lambda$ expression $e$ is syntactically correct in a context $C$, then $C[e]$ is syntactically correct in the context consisting of only the declarations of $C$.

16.5 The Meaning of a Module

I now define the meaning of a module in a context $C$. Modules can appear as submodules of other modules. The meaning of a module depends on a context. For an external module, which is not a submodule of another module, the context consists of declarations and definitions of all the built-in operators of TLA$^+$, plus definitions of some set of modules. This set of modules is discussed in Section 16.6 below.

The meaning of a module consists of six sets:

$Dcl$ A set of declarations. They come from constant and variable declarations and declarations in extended modules (modules appearing in an extends statement).

$GDef$ A set of global definitions. They come from ordinary (non-local) definitions and global definitions in extended and instantiated modules.

$LDef$ A set of local definitions. They come from local definitions and local instantiations of modules. (Local definitions are not obtained by other modules that extend or instantiate the module.)

$MDef$ A set of module definitions. They come from submodules of the module and of extended modules.
16.5. THE MEANING OF A MODULE

Ass A set of assumptions. They come from assume statements and from extended modules.

Thm A set of theorems. They come from theorem statements, from theorems in extended modules, and from the assumptions and theorems of instantiated modules, as explained below.

The only operator symbols appearing in the λ expressions of definitions in GDef and LDef, and in the expressions in Ass and Thm, are declared in either C or in Dcl. More precisely, all of those λ expressions and expressions are syntactically correct in the context consisting of the union of Dcl and the declarations in C.

Only a syntactically correct module has a meaning. A syntactically correct module is also semantically correct iff every theorem in Thm is implied by the assumptions in Ass and the rules of TLA, applied in the context C ∪ Dcl. I won’t explain formally what that means. However, note that the validity of a theorem can depend on whether an operator symbol is a constant or a variable. For example, $c' = c$ is a theorem if $c$ is a constant, but not if it is a variable.\footnote{The formula $c' = c$ asserts that, for any pair of states, the value of $c$ in the first state equals its value in the second state. This is true iff $c$ is a constant, since the value only of a constant is the same in all states.}

The meaning of a module in a context C is defined by an algorithm for computing these six sets. The algorithm computes the six sets by processing each statement in the module in turn, from beginning to end. The meaning of the module is the value of those sets when the end of the module is reached.

Initially, all six sets are empty. The rules for handling each possible type of statement is given below. In these rules, the current context CC is defined to be the union of C, Dcl, GDef, LDef, and MDef.

When the algorithm adds elements to the sets GDef, LDef, Ass, and Thm, it uses α conversion to ensure that no defined or declared operator name appears as a λ parameter. This α conversion is not explicitly mentioned.

**EXTENDS**

An extends statement has the form

\[
\text{extends } M_1, \ldots, M_n
\]

where each $M_i$ is a module name. This statement must be the first one in the module. The statement sets the values of Dcl, GDef, MDef, Ass, and Thm equal to the union of the corresponding values for the module meanings assigned by C to the module names $M_i$.

This statement is legal iff the module names $M_i$ are all defined in C, and the resulting current context CC does not assign more than one meaning to any symbol. It is illegal for one of the modules $M_i$ to define a symbol and another
to declare the same symbol. It is also illegal for the resulting value of \( \mathcal{C} \) to have two different declarations or two different definitions of the same operator name.

Since \( \mathcal{C} \) is a set, by definition it cannot have multiple copies of the same definition or declaration. This means that an operator name can be declared or defined in two different modules \( M_i \), as long as those declarations or definitions are the same. Two declarations are the same if they assign the same level and arity. The semantics of TLA\(^+\) do not specify precisely what it means for two definitions to be the same. A tool may or may not consider the two definitions

\[
A \triangleq \lambda X : \cup (X, Z) \quad A \triangleq \lambda Y : \cup (Y, Z)
\]

to be the same. It does guarantee that two definitions are the same if they come from the same definition statement in the same module. For example, suppose \( M_1 \) and \( M_2 \) both extend the Naturals module. Then modules Naturals, \( M_1 \), and \( M_2 \) all define \( + \). However, all three definitions are the same, because they all come from the statement that provides the definition for Naturals. (That definition might in turn come from another module extended or instantiated by the Naturals module.) Hence, the multiple definitions of \( + \) and other operators on natural numbers obtained by the statement

\[
\text{EXTENDS } \text{Naturals}, M_1, M_2
\]

are legal.

### Declarations

A declaration statement has one of the forms

\[
\text{CONSTANT } c_1, \ldots, c_n \quad \text{VARIABLE } v_1, \ldots, v_n
\]

where the \( v_i \) are operator names and the \( c_i \) are operator names or have the form \( Op(-, \ldots, -) \) for some operator name \( Op \). This statement adds to the set \( \mathcal{D} \) the obvious declarations. It is legal iff none of the declared operators are defined in \( \mathcal{C} \) or are declared in \( \mathcal{C} \). The new current context \( \mathcal{C} \) must not contain multiple declarations for the same operator name. As observed above, \( \mathcal{D} \) is a set, so it cannot contain multiple copies of the same declaration. Hence, multiple equivalent declarations of the same operator name are allowed.

### Operator Definitions

A global operator definition\(^6\) has one of the two forms

\[
Op \triangleq \text{exp} \quad Op(p_1, \ldots, p_n) \triangleq \text{exp}
\]

\(^6\)An operator definition statement should not be confused with a definition clause in a \textit{LET} expression. The meaning of a \textit{LET} expression is described in Section 16.4.
where $Op$ is an operator name, $exp$ is a symbol, and each $p_i$ is either an operator name or has the form $P(\ldots)$ where $P$ is an operator name. We consider the first form an instance of the second with $n = 0$.

This statement if legal if the $\lambda$ expression $\lambda p_1, \ldots, p_n : exp$ is legal in the context $CC$. In particular, no $\lambda$ parameter in this $\lambda$ expression can be defined or declared in $CC$. The statement adds to $GDef$ the definition that assigns to $Op$ the $\lambda$ expression $CC[\lambda p_1, \ldots, p_n : exp]$.

A local operator definition has one of the two forms

\[
\text{LOCAL } Op \overset{\Delta}{=} exp \quad \text{LOCAL } Op(p_1, \ldots, p_n) \overset{\Delta}{=} exp
\]

It is the same as a global definition, except that it adds the definition to $LDef$ instead of $GDef$.

**Function Definitions**

A global function definition has the form

\[
Op[\text{fcnargs}] \overset{\Delta}{=} exp
\]

where $\text{fcnargs}$ is a comma-separated list of elements, each having the form $Id_1, \ldots, Id_n \in S$ or $(Id_1, \ldots, Id_n) \in S$. It is equivalent to the global operator definition

\[
Op \overset{\Delta}{=} \text{choose } Op : Op = [\text{fcnargs} \mapsto exp]
\]

A local function definition, which has the form

\[
\text{LOCAL } Op[\text{fcnargs}] \overset{\Delta}{=} exp
\]

is equivalent to the analogous local operator definition.

**Instantiation**

We consider first a global instantiation of the form:

\[
(16.3) \quad I(p_1, \ldots, p_m) \overset{\Delta}{=} \text{instance } N \text{ with } q_1 \leftarrow e_1, \ldots, q_n \leftarrow e_n
\]

For this to be legal, $N$ must be a module name defined in $CC$. Let $NDcl$, $NDef$, $NAss$, and $NThm$ be the sets $Dcl$, $GDef$, $Ass$, and $Thm$ in the meaning assigned to $N$ by $CC$. The $q_i$ must be distinct symbols declared by $NDcl$. We add a WITH clause of the form $Op \leftarrow Op$ for any operator symbol $Op$ that is declared in $NDcl$ but is not one of the $q_i$, so the $q_i$ constitute all the symbols declared in $NDcl$.

Neither $I$ nor any of the definition parameters (the operator names in the $p_i$) may be defined or declared in $CC$. Let $D$ be the context obtained by adding to $CC$ the obvious constant-level declaration for each $p_i$. Then $e_i$ must be syntactically
correct in the context $D$, and $D[e_i]$ must have the same arity as $q_i$, for each $i \in 1..n$.

There is one further legality condition, which depends on the nature of $N$. Module $N$ is a constant module if and only if every declaration in $NDcl$ has constant level, and every operator appearing in every definition in $NDef$ has constant level. If $N$ is not a constant module, then the following condition must be satisfied for each $i$:

- If $q_i$ is declared in $NDcl$ to be a constant operator, then $D[e_i]$ has constant level.
- If $q_i$ is declared in $NDcl$ to be a variable (a 0th-order operator of level 1), then $D[e_i]$ has level 0 or 1.

The reason for this condition is discussed in Section 16.7 below.

For any $\lambda$ expression $e$, let $\tau$ be the expression obtained from $e$ by substituting $e_i$ for $q_i$, for all $i \in 1..n$. For each definition $Op \triangleq \lambda r_1, \ldots, r_p : e$ in $NDef$, the definition

$$I! Op \triangleq \lambda p_1, \ldots, p_m, r_1, \ldots, r_p : \tau$$

is added to $GDef$. (Remember that $I! Op(c_1, \ldots, c_m, d_1, \ldots, d_n)$ is actually written in TLA$^+$ as $I(c_1, \ldots, c_m)! Op(d_1, \ldots, d_n)$.) There is a further subtle point in the definition of the substitution that yields $\tau$; it is described in Section 16.7 below.

Also added to $GDef$ is the special definition $I \triangleq ?$. (This prevents $I$ from later being defined or declared as an operator name.)

If $NAss$ equals the set $\{a_1, \ldots, a_k\}$ of assumptions, then for each $t_j$ in $NThm$, we add to $Thm$ a theorem asserting that the assumptions $\overline{a_i}$ imply $\overline{t_j}$. (Because this theorem may contain the parameters $p_i$, it cannot in general be expressed in the subset of TLA$^+$ used for writing specifications.)

A global instance statement can also have the two forms:

$$I \triangleq \text{instance } N \text{ with } q_1 \leftarrow e_1, \ldots, q_n \leftarrow e_n$$

$$\text{instance } N \text{ with } q_1 \leftarrow e_1, \ldots, q_n \leftarrow e_n$$

The first is just the $m = 0$ case of (16.3); the second is similar to the first, except the definitions added to $GDef$ do not have $I!$ prepended to the operator names. In all three forms of the statement, the with clause can be omitted; this is equivalent to the case $n = 0$ of these statements. (Remember that all the parameters of module $N$ must be either explicitly or implicitly instantiated.)

A local instance statement consists of the keyword local followed by an instance statement of the form described above. It is handled in a similar fashion to a global instance statement, except that all definitions are added to $LDef$ instead of $GDef$. 
16.5.1 Theorems and Assumptions

A theorem has one of the forms

\[
\text{THEOREM } exp \quad \text{THEOREM } Op \triangleq exp
\]

where \( exp \) is an expression, which must be legal in the current context \( CC \). The first form adds the theorem \( CC[exp] \) is added to the set \( Thm \). The second form is equivalent to the pair of statements:

\[
Op \triangleq exp \\
\text{THEOREM } exp
\]

An assumption has one of the forms

\[
\text{ASSUME } exp \quad \text{ASSUME } Op \triangleq exp
\]

It is similar to a theorem except that \( CC[exp] \) is added to the set \( Ass \).

16.5.2 Submodules

A module can contain a submodule, which is a complete module that begins with

```
MODULE N
```

for some module name \( N \), and ends with

```
```

This is legal iff the module name \( N \) is not defined in \( CC \) and the module is legal in the context \( CC \). In this case, the module definition that assigns to \( N \) the meaning of the submodule in context \( CC \) is added to \( MDef \).

A submodule can be used in an \textit{instance} statement within the current module, or within a module that extends the current module.

This completes the definition of the meaning of a module.

16.6 Finding Modules

Section 16.5 above defines the meaning of a module in a context \( C \) that contains a module definition for every module name appearing in an \textit{extends} or \textit{instance} statement that is not the name of submodule of the module itself.

In practice, a tool (or a person) begins interpreting a module \( M \) in a context \( C_0 \) containing only declarations and definitions of the built-in TLA\(^+\) operator names. When an \textit{extends} or \textit{instance} statement is encountered that mentions a module named \( N \) that is not defined in the current context \( CC \) of \( M \), the
tool finds the module named $N$, interprets it in the context $C_0$, and then adds the module definition for $N$ to $C_0$ and to $CC$.

The definition of the TLA$^+$ language does not specify how a tool finds a module named $N$. The tool will most likely look for the module in a file whose name is derived in some standard way from $N$.

### 16.7 The Semantics of Instantiation

The meaning of an instance statement is defined above, in Section 16.5. Instantiation is the formalization in TLA$^+$ of the mathematical process of substitution. A fundamental property of mathematics is that substitution in a valid formula yields a valid formula. As we saw above, this property is asserted by the meaning of TLA$^+$ instantiation. If module $N$ is semantically correct, so its theorems follow from its assumptions, then the instantiated versions of its theorems follow from the instantiated versions of its assumptions.

For simplicity, let’s suppose module $N$ has no assumptions and contains

$$T \triangleq \ldots$$

\text{THEOREM } T

and module $M$ contains the statement

$$I \triangleq \text{instance } N \text{ with } \ldots$$

Preserving validity under instantiation means that the assertion

\text{THEOREM } I!T

in module $M$ is valid.

The level rule for instantiation is necessary for instantiation to preserve validity. Recall that this rule requires that, when instantiating a nonconstant module, a constant parameter can be instantiated with only a constant-level expression or operator. To see why this rule is necessary, let $c$ be a declared constant in module $N$, let $T$ be $c' = c$, and let the instance statement be

$$I \triangleq \text{instance } N \text{ with } c \leftarrow x$$

where $x$ is declared in $M$ to be a variable. Then $I!T$ equals $x' = x$, which is not a valid formula. (There exists a pair of states in which $x$ has different values.)

Certain nonconstant operators of TLA$^+$ make preserving validity under instantiation tricky. Most notable of these operators is Enabled. Suppose $x$ and $y$ are declared variables of module $N$, and $T$ is defined by

$$T \triangleq \text{Enabled } (x' = 0 \land y' = 1)$$
Then \( T \) is equivalent to \( \text{true} \), so it is a theorem of module \( N \). (For any state \( s \), there exists a state \( t \) in which \( x = 0 \) and \( y = 1 \).) Now suppose \( z \) is a declared variable of module \( M \), and consider the instantiation

\[
I \triangleq \text{instance } N \text{ with } x \leftarrow z, \; y \leftarrow z
\]

With naive substitution, \( I \! T \) would equal

\[
\text{Enabled}(z' = 0 \land z' = 1)
\]

which is equivalent to \( \text{false} \). (For any state \( s \), there is no state \( t \) in which \( z = 0 \) and \( z = 1 \) are both true.) Hence, \( I \! T \) would not be a theorem, so instantiation would not preserve validity.

Naive substitution in a formula of the form \( \text{Enabled} \; A \) does not preserve validity because the primed variables in \( A \) are really bound variables. The formula \( \text{Enabled} \; A \) asserts that there exist values of the primed variables such that \( A \) is true. As logicians know, when substituting in an expression with quantifiers, one does not substitute for bound occurrences of an identifier. This problem doesn’t arise with quantifiers in TLA\(^+\) because its rules do not allow the same identifier to occur both bound and free in an expression. However, this can’t be avoided in the expression \( \text{Enabled} \; A \) because the quantification is implicit.

Recall that in defining the meaning of the statement

\[
I(p_1, \ldots, p_m) \triangleq \text{instance } N \text{ with } q_1 \leftarrow e_1, \ldots, q_n \leftarrow e_n
\]

I defined \( \overline{\pi} \) to be the expression obtained by substituting \( e_i \) for \( q_i \) in \( e \), for \( i \in 1 \ldots n \). When performing this substitution, for each subexpression of \( e \) of the form \( \text{Enabled} \; A \) and each declared variable \( q \) of module \( N \), we replace every primed occurrence of \( q \) in \( A \) with a new symbol, which I will write \( \$q \), that does not appear in \( A \). This new symbol is considered to be bound by the \( \text{Enabled} \) operator. For example, the module

\[
\begin{align*}
\text{MODULE } N \\
\text{VARIABLE } u \\
G(v, A) & \triangleq \text{Enabled} \; (A \lor (\{u, v\}' = \{u, v\})) \\
H & \triangleq (u' = u) \land G(u, u' \neq u)
\end{align*}
\]

has as its global definitions the set:

\[
\{ \begin{align*}
G & \triangleq \lambda v, A : \text{Enabled} \; (A \lor (\{u, v\}' = \{u, v\})) \\
H & \triangleq (u' = u) \land \text{Enabled} \; ((u' \neq u) \lor (\{u, u\}' = \{u, u\}))
\end{align*} \}
\]

The statement

\[
I \triangleq \text{instance } N \text{ with } u \leftarrow x
\]
adds the following definitions to the current module:

\[ I!G \triangleq \lambda v, A : \text{ENABLED} \ (A \lor \{u, v\}' = \{u, v\}) \]
\[ I!H \triangleq (x' = x) \land \text{ENABLED} \ ((u' \neq x) \lor ((u, \$u)' = \{x, x\})) \]

Observe that even though \( H \) equals \( G(u, u' \neq u) \) in module \( N \), and the instantiation substitutes \( x \) for \( u \), the instantiated formula \( I!H \) does not equal \( I!G(x, x' \neq x) \).

As another example, consider the module

```
VARIABLES u, v
A \triangleq (u' = u) \land (v' \neq v)
B(d) \triangleq \text{ENABLED} \ d
C \triangleq B(A)
```

The instantiation

\[ I \triangleq \text{INSTANCE} \ N \ \text{WITH} \ u \leftarrow x, \ v \leftarrow x \]

adds the following definitions to the current module

\[ I!A \triangleq (x' = x) \land (x' \neq x) \]
\[ I!B \triangleq \lambda d : \text{ENABLED} \ d \]
\[ I!C \triangleq \text{ENABLED} \ ((u' = x) \land (v' \neq x)) \]

Observe that \( I!C \) is not equivalent to \( I!B(I!A) \). In fact, \( I!C \equiv \text{TRUE} \) and \( I!B(I!A) \equiv \text{FALSE} \).

We say that instantiation \textit{distributes} over an operator \( Op \) if

\[ Op(e_1, \ldots, e_n) = Op(e_1', \ldots, e_n') \]

for any expressions \( e_i \), where the overlining operator (\( \overline{\text{\}} \)) denotes some arbitrary instantiation. Instantiation distributes over all constant operators—for example, \(+\), \(\subseteq\), and \(\exists\). (Instantiation distributes over a quantifier because TLA\(^+\) does not permit an instantiation that substitutes for the quantifier’s bound variables.) Instantiation also distributes over most of the nonconstant operators of TLA\(^+\), like priming (\( ' \)) and \(\Box\).

If an operator \( Op \) implicitly binds some identifiers in its arguments, then instantiation would not preserve validity if it distributed over \( Op \). Our rules for instantiating in an \text{ENABLED} expression imply that instantiation does not distribute over \text{ENABLED}. It also does not distribute over any operator defined in terms of \text{ENABLED}—in particular, the built-in operators \(\text{WF} \) and \(\text{SF} \).

There are two other TLA\(^+\) operators that implicitly bind identifiers: the action composition operator \(\cdot\), defined in Section 15.2.2, and the temporal
while operator \( \Rightarrow \), introduced in Section 10.2. The rule for instantiating an expression \( A \cdot B \) is similar to that for ENABLED \( A \)—namely, bound occurrences of variables are replaced by a new symbol. In the expression \( A \cdot B \), primed occurrences of variables in \( A \) and unprimed occurrences in \( B \) are bound. The rule for \( \Rightarrow \) is that, when instantiating an expression \( F \Rightarrow G \), that expression is replaced by its definition. To define \( F \Rightarrow G \), let \( x \) be the tuple \( (x_1, \ldots, x_n) \) of all declared variables; let \( b, \widehat{x}_1, \ldots, \widehat{x}_n \) be symbols distinct from the \( x_i \) and from any bound identifiers in \( F \) or \( G \); and let \( \hat{e} \) be the expression obtained from an expression \( e \) by substituting the variables \( \widehat{x}_i \) for the corresponding variables \( x_i \). Then \( F \Rightarrow G \) is defined to equal

\[
\forall b : ( \land (b = \text{TRUE}) \land \Box [b' = \text{FALSE}]_b \\
\land \exists \widehat{x}_1, \ldots, \widehat{x}_n : \widehat{F} \land \Box (b \Rightarrow (x = \widehat{x}))) \\
\Rightarrow \exists \widehat{x}_1, \ldots, \widehat{x}_n : \widehat{G} \land (x = \widehat{x}) \land \Box [b \Rightarrow (x' = \widehat{x}')]_{(b,x,\widehat{x})}
\]
Chapter 17

The Standard Modules

We provide several standard modules for use in TLA+ specifications. Some of the definitions they contain are subtle—for example, the definitions of the set of real numbers and its operators. Others, such as the definition of $1 \ldots n$, are obvious. There are two reasons to use standard modules. First, specifications are easier to read when they use basic operators that we’re already familiar with. Second, tools can have built-in knowledge of standard operators. For example, the TLC model checker (Chapter 13) has efficient implementations of some standard modules; and a theorem-prover might implement special decision procedures for some standard operators.

17.1 Module Sequences

The Sequences module was introduced in Section 4.1 on page 35. Most of the operators it defines have already been explained. The exceptions are:

$\text{SubSeq}(s, m, n)$ The subsequence $\langle s[m], s[m + 1], \ldots, s[n] \rangle$ consisting of the $m^{th}$ through $n^{th}$ elements of $s$. It is undefined if $m < 1$ or $n > \text{Len}(s)$, except that it equals the empty sequence if $m > n$.

$\text{SelectSeq}(s, \text{test})$ The subsequence of $s$ consisting of the elements $s[i]$ such that $\text{test}[s[i]]$ equals TRUE. For example:

$$\text{PosSubSeq}(s) \triangleq \text{LET IsPos}(n) \triangleq n > 0 \text{ IN } \text{SelectSeq}(s, \text{IsPos})$$

defines $\text{PosSubSeq}((0, 3, -2, 5))$ to equal $\langle 3, 5 \rangle$.

The Sequences module uses operators on natural numbers, so we might expect it to extend the Naturals module. However, this would mean that any module
CHAPTER 17. THE STANDARD MODULES

MODULE Sequences

Defines operators on finite sequences, where a sequence of length $n$ is represented as a function whose domain is the set $1 .. n$ (the set \{1, 2, ..., $n$\}). This is also how TLA+ defines an $n$-tuple, so tuples are sequences.

<table>
<thead>
<tr>
<th>Sequence operator</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$Seq(S)$</td>
<td>$\triangleq$</td>
</tr>
<tr>
<td>$Len(s)$</td>
<td>$\triangleq$</td>
</tr>
<tr>
<td>$s \circ t$</td>
<td>$\triangleq$</td>
</tr>
<tr>
<td>$\text{Append}(s, e)$</td>
<td>$\triangleq$</td>
</tr>
<tr>
<td>$\text{Head}(s)$</td>
<td>$\triangleq$</td>
</tr>
<tr>
<td>$\text{Tail}(s)$</td>
<td>$\triangleq$</td>
</tr>
<tr>
<td>$\text{SubSeq}(s, m, n)$</td>
<td>$\triangleq$</td>
</tr>
<tr>
<td>$\text{SelectSeq}(s, \text{test}(_))$</td>
<td>$\triangleq$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{LET } F[i \in 0 .. \text{Len}(s)] \overset{\Delta}{=} \text{IF } i = 0 \text{ THEN } { } \text{ ELSE IF } \text{test}(s[i]) \text{ THEN } \text{Append}(F[i - 1], s[i]) \text{ ELSE } F[i - 1] \text{ IN } F[\text{Len}(s)]$</td>
</tr>
</tbody>
</table>

Figure 17.1: The standard Sequences module.

that extends Sequences would then also extend Naturals. Just in case someone wants to use sequences without extending the Naturals module, the Sequences module contains the statement:

LOCAL INSTANCE Naturals

This statement introduces the definitions from the Naturals module, just as an ordinary instance statement would, but it does not export those definitions to another module that extends or instantiates the Sequences module. The local modifier can also precede an ordinary definition; it has the effect of making that definition usable within the current module, but not in a module that extends or instantiates it. (The local modifier cannot be used with parameter declarations.)

Everything else that appears in the Sequences module should be familiar. The module is in Figure 17.1 on this page.
17.2 Module $\textit{FiniteSets}$

As described in Section 6.1 on page 66, the $\textit{FiniteSets}$ module defines the two operators $\textit{IsFiniteSet}$ and $\textit{Cardinality}$. The definition of $\textit{Cardinality}$ is discussed on pages 69–70. The module itself is in Figure 17.2 on this page.

17.3 Module $\textit{Bags}$

A bag, also called a multiset, is a set that can contain multiple copies of the same element. A bag can have infinitely many elements, but only finitely many copies of any single element. Bags are sometimes useful for representing data structures. For example, the state of a network in which messages can be delivered in any order may be represented as a bag of messages in transit. Multiple copies of an element in the bag represent multiple copies of the same message in transit.

The $\textit{Bags}$ module defines a bag to be a function whose range is a subset of the positive integers. An element $e$ belongs to bag $B$ iff $e$ is in the domain of $B$, in which case bag $B$ contains $B[e]$ copies of $e$. The module defines the following operators. In our customary style, we leave unspecified the value obtained by applying an operator on bags to something other than a bag.

- $\textit{IsABag}(B)$ True iff $B$ is a bag.
- $\textit{BagToSet}(B)$ The set of elements at least one copy of which are in the bag $B$.
- $\textit{SetToBag}(S)$ The bag that contains one copy of every element in the set $S$.
- $\textit{BagIn}(e, B)$ True iff bag $B$ contains at least one copy of $e$. $\textit{BagsIn}$ is the $\in$ operator for bags.
EmptyBag

The bag containing no elements.

CopiesIn\((e, B)\)

The number of copies of \(e\) in bag \(B\). If \(BagIn(e, B)\) is false, then \(CopiesIn(e, B) = 0\).

\(B1 \oplus B2\)

The union of bags \(B1\) and \(B2\). The operator \(\oplus\) satisfies

\[CopiesIn(e, B1 \oplus B2) = CopiesIn(e, B1) + CopiesIn(e, B2)\]

for any \(e\) and any bags \(B1\) and \(B2\).

\(B1 \ominus B2\)

The bag \(B1\) with the elements of \(B2\) removed—that is, with one copy of an element removed from \(B1\) for each copy of the same element in \(B2\). If \(B2\) has at least as many copies of \(e\) as \(B1\), then \(B1 \ominus B2\) has no copies of \(e\).

BagUnion\((S)\)

The bag union of all elements of the set \(S\) of bags. For example, \(BagUnion\{B1, B2, B3\}\) equals \(B1 \oplus B2 \oplus B3\).

\(B1 \sqsubseteq B2\)

True iff, for all \(e\), bag \(B2\) has at least as many copies of \(e\) as bag \(B1\) does. Thus, \(\sqsubseteq\) is the analog for bags of \(\subseteq\).

SubBag\((B)\)

The set of all subbags of bag \(B\). SubBag is the bag analog of the subset operator.

BagOfAll\((F, B)\)

The bag analog of the construct \(\{F(x) : x \in B\}\). It is the bag that contains, for each element \(e\) of bag \(B\), one copy of \(F(e)\) for every copy of \(e\) in \(B\). This defines a bag iff, for any value \(v\), the set of \(e\) in \(B\) such that \(F(e) = v\) is finite.

BagCardinality\((B)\)

If \(B\) is a finite bag (one such that \(BagToSet(B)\) is a finite set), then this is its cardinality—the total number of copies of elements in \(B\). Its value is unspecified if \(B\) is not a finite bag.

The module appears in Figure 17.3 on the next page. Note the local definition of \(Sum\), which makes \(Sum\) defined within the \(Bags\) module but not in any module that extends or instantiates it.

17.4 Module \(RealTime\)

The \(RealTime\) module will be described in Chapter 9.
LOCAL INSTANCE Naturals Import definitions from Naturals, but don’t export them.

IsABag(B) \triangleq B \in \{n \in Nat : n > 0\} \quad \text{True iff } B \text{ is a bag.}

BagToSet(B) \triangleq \text{DOMAIN } B \quad \text{The bag union of all elements of the set.}

SetToBag(S) \triangleq [e \in S \mapsto 1] \quad \text{The bag that contains one copy of every element of the set } S.

BagIn(e, B) \triangleq e \in \text{BagToSet}(B) \quad \text{The } \in \text{ operator for bags.}

EmptyBag \triangleq \text{SetToBag}() \quad \text{The bag analog of the set.}

B1 \oplus B2 \triangleq \text{The set of bags } B1 \text{ and } B2.

\begin{align*}
[e \in (\text{DOMAIN } B1) \cup (\text{DOMAIN } B2) & \mapsto \\
(\text{IF } e \in \text{DOMAIN } B1 & \text{ THEN } B1[e] \text{ ELSE } 0) + (\text{IF } e \in \text{DOMAIN } B2 & \text{ THEN } B2[e] \text{ ELSE } 0)]
\end{align*}

\begin{align*}
B1 \ominus B2 \triangleq \text{The bag } B1 \text{ with the elements of } B2 \text{ removed.}
\end{align*}

\begin{align*}
\text{LET } B \triangleq [e \in \text{DOMAIN } B1 & \mapsto \text{IF } e \in \text{DOMAIN } B2 \text{ THEN } B1[e] - B2[e] \text{ ELSE } B1[e]] \\
\text{IN } [e \in \{d \in \text{DOMAIN } B : B[d] > 0\} & \mapsto B[e]]
\end{align*}

LOCAL \text{ Sum}(f) \triangleq \text{The sum of } f[x]\text{ for all } x \text{ in } \text{DOMAIN } f.

\begin{align*}
\text{LET } D\text{Sum}\{S \in \subset \text{DOMAIN } f\} & \triangleq \text{LET } elt \triangleq \text{CHOOSE } e \in S : \text{TRUE} \\
\text{IN } \text{IF } S = \{\} & \text{ THEN } 0 \\
\text{ELSE } f[elt] + D\text{Sum}\{S \setminus \{elt\}\}
\end{align*}

\begin{align*}
\text{IN } D\text{Sum}[\text{DOMAIN } f]
\end{align*}

\begin{align*}
\text{BagUnion}(S) & \triangleq \text{The bag union of all elements of the set } S \text{ of bags.} \\
[e \in \text{UNION } \{\text{BagToSet}(B) : B \in S\} & \mapsto \text{Sum}\{B \in S \mapsto \text{IF } \text{BagIn}(e, B) \text{ THEN } B[e] \text{ ELSE } 0\}]
\end{align*}

\begin{align*}
B1 \subseteq B2 \triangleq & \quad \text{AND } (\text{DOMAIN } B1) \subseteq (\text{DOMAIN } B2) \quad \text{The subset operator for bags.} \\
\text{AND } \forall e \in \text{DOMAIN } B1 & \quad B1[e] \leq B2[e]
\end{align*}

\begin{align*}
\text{SubBag}(B) & \triangleq \text{The set of all subbags of bag } B. \\
\text{LET } \text{AllBagsOfSubset} & \triangleq \text{The set of bags } SB \text{ such that } \text{BagToSet}(SB) \subseteq \text{BagToSet}(B). \\
\text{UNION } \{SB \mapsto \{n \in Nat : n > 0\} & \mapsto \text{SUM}\{B \in S \mapsto \text{IF } \text{BagIn}(e, B) \text{ THEN } B[e] \text{ ELSE } 0\}\}
\end{align*}

\begin{align*}
\text{BagOfAll}(F(\_), B) & \triangleq \text{The bag analog of the set } \{F(x) : x \in B\} \text{ for a set } B. \\
[e \in \{F(d) : d \in \text{BagToSet}(B)\} & \mapsto \\
\text{SUM}\{d \in \text{BagToSet}(B) \mapsto \text{IF } F(d) = e \text{ THEN } B[d] \text{ ELSE } 0\}\}
\end{align*}

\begin{align*}
\text{BagCardinality}(B) & \triangleq \text{SUM}(B) \quad \text{the total number of copies of elements in bag } B. \\
\text{CopiesIn}(e, B) & \triangleq \text{IF } \text{BagIn}(e, B) \text{ THEN } B[e] \text{ ELSE } 0 \quad \text{The number of copies of } e \text{ in } B.
\end{align*}

Figure 17.3: The standard Bags module.
17.5 The Numbers Modules

The usual sets of numbers and operators on them are defined in the three modules Naturals, Integers, and Reals. These modules are tricky because their definitions must be consistent. A module \( M \) might extend both the Naturals module and another module that extends the Reals module. The module \( M \) thereby obtains two definitions of an operator such as \(+\), one from Naturals and one from Reals. These two definitions of \(+\) must be the same. To make them the same, we have them both come from the definition of \(+\) in a module ProtoReals, which is locally instantiated by both Naturals and Reals.

The Naturals module defines the following operators:

- \( + \): addition
- \( * \): multiplication
- \( < \): less than
- \( \leq \): less than or equal
- \( \geq \): greater than or equal
- \( > \): greater than
- \( \div \): integer division
- \( \% \): modulus

Except for \( \div \), these operators are all either standard or explained in Chapter 2. We define integer division \( \div \) and modulus \( \% \) so that for any integer \( a \) and positive integer \( b \):

\[
a \% b = 0 \ldots (b - 1) \quad a = b \times (a \div b) + (a \% b)
\]

The Integers module extends the Naturals module and also defines the set \( \text{Int} \) of integers and unary minus (\( - \)). The Reals module extends Integers and introduces the set \( \text{Real} \) of real numbers and ordinary division (\( / \)). In mathematics, (unlike programming languages), integers are real numbers. Hence, \( \text{Nat} \) is a subset of \( \text{Int} \), which is a subset of \( \text{Real} \).

The Reals module also defines the special value \( \text{Infinity} \). Infinity, which represents a mathematical \( \infty \), satisfies the following two properties:

\[
\forall r \in \text{Real} : -\text{Infinity} < r < \text{Infinity} \quad -(-\text{Infinity}) = \text{Infinity}
\]

The precise details of the number modules are of no practical importance. When writing specifications, you can just assume that the operators they define have their usual meanings. If you want to prove something about a specification, you can reason about numbers however you want. Tools like model checkers and theorem provers that care about these operators will have their own ways of handling them. The modules are given here mainly for completeness. They can also serve as models if you want to define other basic mathematical structures. However, such definitions are rarely necessary for engineering specifications.

The set \( \text{Nat} \) of natural numbers, with its zero element and successor function is defined in the Peano module, which appears in Figure 17.4 on the next page. It simply defines the naturals to be a set satisfying Peano’s axioms \([\text{?}]\). This definition is separated into its own module for the following reason. As explained in Section 15.1.7 (page 195) and Section 15.1.8 (page 195), the meanings of tuples and strings are defined in terms of the natural numbers. The Peano module,
17.5. THE NUMBERS MODULES

MODULE Peano

This module defines \( \text{Nat} \) to be an arbitrary set satisfying Peano’s axioms with zero element \( \text{Zero} \) and successor function \( \text{Succ} \). It does not use strings or tuples, which in TLA\(^+\) are defined in terms of natural numbers.

\[
\begin{align*}
PeanoAxioms(N, Z, Sc) & \triangleq \\
& \quad \land Z \in N \\
& \quad \land Sc \in [N \to N] \\
& \quad \land \forall n \in N : (\exists m \in N : n = Sc[m]) \equiv (n \neq Z) \\
& \quad \land \forall S \in \text{subset } N : (Z \in S) \land (\forall n \in S : Sc[n] \in S) \Rightarrow (S = N)
\end{align*}
\]

ASSUME \( \exists N, Z, Sc : PeanoAxioms(N, Z, Sc) \)  
Asserts the existence of a set satisfying Peano’s axioms.

\[
\begin{align*}
\text{Succ} & \triangleq \text{CHOOSE } Sc : \exists N, Z : PeanoAxioms(N, Z, Sc) \\
\text{Nat} & \triangleq \text{DOMAIN Succ} \\
\text{Zero} & \triangleq \text{CHOOSE } Z : PeanoAxioms(Nat, Z, Succ)
\end{align*}
\]

Figure 17.4: The Peano module.

which defines the natural numbers, does not use tuples or strings. Hence, there is no circularity.

Most of the definitions in the number modules come from module \textit{ProtoReals} in Figures 17.5 and 17.6 on the following two pages. To define the real numbers, it uses the well-known mathematical result that the reals are uniquely defined, up to isomorphism, as an ordered field in which every subset bounded from above has a least upper bound. The details will be of interest only to mathematically sophisticated readers who are curious about the formalization of ordinary mathematics. I hope that those readers will be as impressed as I am by how easy this formalization is—once you understand the mathematics.

Given the \textit{ProtoReals} module, the rest is easy. The \textit{Naturals}, \textit{Integers}, and \textit{Reals} modules appear in Figures 17.7–17.9 on page 228. Perhaps the most striking thing about them is the ugliness of an operator like \( R^+ \), which is the version of + obtained by instantiating \textit{ProtoReals} under the name \( R \). It demonstrates that you should not use infix operators when writing a module that may be used with a named instantiation.
This module provides the basic definitions for the Naturals, Integers, and Reals module. It does this by defining the real numbers to be a complete ordered field containing the naturals.

EXTENDS Peano

IsModelOfReals(R, Plus, Times, Leq) ≜

Asserts that $R$ satisfies the properties of the reals with $a + b = \text{Plus}[a, b]$, $a \cdot b = \text{Times}[a, b]$, and $(a \leq b) = (\langle a, b \rangle \in \text{Leq})$. (We will have to quantify over the arguments, so they must be values, not operators.)

LET IsAbelianGroup(G, Id, +, -) ≜

Asserts that $G$ is an Abelian group with identity $\text{Id}$ and group operation $\text{+.}$. The first two conjuncts assert that $\text{Nat}$ is embedded in $R$. The next three conjuncts assert that $R$ is a field. The next two conjuncts assert that $R$ is an ordered field. The last conjunct asserts that every subset $S$ of $R$ bounded from above has a least upper bound $\text{sup}$. Theorem 17.5: The $\text{ProtoReals}$ module (beginning).
We will define \( \text{Infinity} \), \(<\), and \(-\) so \(-\text{Infinity} < r < \text{Infinity}\), for any \( r \in \text{Real} \), and \(-(-\text{Infinity}) = \text{Infinity}\).

\[
\text{Infinity} \triangleq \text{choose } x : x \notin \text{Real}
\]

\[
\text{MinusInfinity} \triangleq \text{choose } x : x \notin \text{Real} \cup \{\text{Infinity}\}
\]

\[
a + b \triangleq \text{RM.Plus}[a, b]
\]

\[
a * b \triangleq \text{RM.Times}[a, b]
\]

\[
a \leq b \triangleq \text{case } \begin{cases} (a \in \text{Real}) \land (b \in \text{Real}) & \rightarrow \langle a, b \rangle \in \text{RM.Leq} \\ (a = \text{Infinity}) \land (b \in \text{Real} \cup \{\text{MinusInfinity}\}) & \rightarrow \text{FALSE} \\ (a \in \text{Real} \cup \{\text{MinusInfinity}\}) \land (b = \text{Infinity}) & \rightarrow \text{TRUE} \\ a = b & \rightarrow \text{TRUE} \end{cases}
\]

\[
a - b \triangleq \text{case } \begin{cases} (a \in \text{Real}) \land (b \in \text{Real}) & \rightarrow \text{choose } c \in \text{Real} : c + b = a \\ (a \in \text{Real}) \land (b = \text{Infinity}) & \rightarrow \text{MinusInfinity} \\ (a \in \text{Real}) \land (b = \text{MinusInfinity}) & \rightarrow \text{Infinity} \end{cases}
\]

\[
a/b \triangleq \text{choose } c \in \text{Real} : a = b * c
\]

\[
\text{Int} \triangleq \text{Nat} \cup \{\text{Zero} - n : n \in \text{Nat}\}
\]

We define \( a^b \) (exponentiation) for \( a > 0 \), \( a \neq 0 \) and \( b \in \text{Int} \), or \( a = 0 \) and \( b > 0 \) by the four axioms:

\[
a^1 = a \\
a^{m+n} = a^m \times a^n \text{ if } a \neq 0 \text{ and } m, n \in \text{Int} \\
a^0 = 0 \text{ if } b > 0 \\
a^{b+c} = a^b \text{ if } a > 0
\]

plus the continuity condition that \( 0 < a \) and \( 0 < b \leq c \) imply \( a^b \leq a^c \).

\[
a^b \triangleq \text{let } \text{RPos} \triangleq \{ r \in \text{Real} \setminus \{\text{Zero}\} : \text{Zero} \leq r \} \\
\text{exp} \triangleq \text{choose } f \in [(\text{RPos} \times \text{Real}) \cup ((\text{Real} \setminus \{0\}) \times \text{Int}) \cup (\{0\} \times \text{RPos}) \rightarrow \text{Real}] : \\
\land \forall r \in \text{Real} : \land f[r, \text{Succ}][\text{Zero}] = r \\
\land \forall m, n \in \text{Int} : f[r, m + n] = f[r, m] * f[r, n] \\
\land \forall r \in \text{RPos} : \land f[\text{Zero}, r] = 0 \\
\land \forall s, t \in \text{Real} : f[r, s + t] = f[f[r, s], t] \\
\land \forall s, t \in \text{RPos} : (s \leq t) \Rightarrow (f[r, s] \leq f[r, t])
\]

\[
\text{IN exp}[a, b]
\]

Figure 17.6: The ProtoReals module (end).
CHAPTER 17. THE STANDARD MODULES

### MODULE Naturals

**LOCAL** $R \triangleq \text{INSTANCE ProtoReals}

**Nat** $\triangleq R!Nat$

$\forall a, b : R!Nat$,

- $a + b \triangleq (a + R! b)$
- $a - b \triangleq (a - R! b)$
- $a * b \triangleq (a * R! b)$
- $a^b \triangleq (a ^ R! b)$

$a \leq b \triangleq (a \leq b) \land (a \neq b)$

$a < b \triangleq (a < b) \land (i \leq b)$

$a > b \triangleq (a > b)$

$a \ldots b \triangleq \{ i \in Nat : (a \leq i) \land (i \leq b) \}$

$a \div b \triangleq \text{CHOOSE } n \in R!Int : \exists r \in 0 \ldots (b - 1) : a = b * n + r$

$a \% b \triangleq a - b * (a \div b)$

---

**Figure 17.7:** The standard Naturals module.

### MODULE Integers

**EXTENDS** Naturals

The Naturals module already defines operators like $+$ to work on all real numbers.

**LOCAL** $R \triangleq \text{INSTANCE ProtoReals}

**Int** $\triangleq R!Int$

$-,- a \triangleq 0-a$ \(\text{Unary } - \text{ is written } \cdot, \text{ when being defined or used as an operator argument.}\)

---

**Figure 17.8:** The standard Integers module.

### MODULE Reals

**EXTENDS** Integers

The Integers module already defines operators like $+$ to work on all real numbers.

**LOCAL** $R \triangleq \text{INSTANCE ProtoReals}

**Real** $\triangleq R!Real$

$a/b \triangleq (a / R! b)$ \(R!/ \text{ is the operator } / \text{ defined in module ProtoReals.}\)

Infinity $\triangleq R!Infinity$

---

**Figure 17.9:** The standard Reals module.
Part V

Appendix
Appendix A

The ASCII Specifications

A.1 The Asynchronous Interface

------------------ MODULE AsynchInterface ------------------
EXTENDS Naturals
CONSTANT Data
VARIABLES val, rdy, ack

TypeInvariant == /
val \in Data
/ rdy \in \{0, 1\}
/ ack \in \{0, 1\}

--------------------------------------------------------------
Init == /
val \in Data
/ rdy \in \{0, 1\}
/ ack = rdy

Send == /
rdy = ack
/ val' \in Data
/ rdy' = 1 - rdy
/ UNCHANGED ack

Rcv == /
rdy # ack
/ ack' = 1 - ack
/ UNCHANGED <<val, rdy>>

Next == Send \/ Rcv

Spec == Init \[\] [Next].<<val, rdy, ack>>

--------------------------------------------------------------
THEOREM Spec => []TypeInvariant

--------------- MODULE Channel -------------
EXTENDS Naturals
CONSTANT Data
VARIABLE chan

TypeInvariant ==
   chan \in \{val : Data, rdy : \{0, 1\}, ack : \{0, 1\}\}

Init == /
   TypeInvariant
   /
   chan.ack = chan.rdy

Send(d) == /
   chan.rdy = chan.ack
   /
   chan' = [chan EXCEPT !.val = d, !.rdy = 1 - @]

Rcv == /
   chan.rdy # chan.ack
   /
   chan' = [chan EXCEPT !.ack = 1 - @]

Next == (E d \in Data : Send(d)) \or Rcv

Spec == Init /
   []\[Next\]_chan

THEOREM Spec => []TypeInvariant

--------------- MODULE InnerFIFO -------------
EXTENDS Naturals, Sequences
CONSTANT Message
VARIABLES in, out, q
InChan == INSTANCE Channel WITH Data <- Message, chan <- in
OutChan == INSTANCE Channel WITH Data <- Message, chan <- out

Init == /
   InChan!Init
   /
   OutChan!Init
   /
   q = << >>

TypeInvariant == /
   InChan!TypeInvariant

A.2 A FIFO

--------------- MODULE InnerFIFO -------------
EXTENDS Naturals, Sequences
CONSTANT Message
VARIABLES in, out, q
InChan == INSTANCE Channel WITH Data <- Message, chan <- in
OutChan == INSTANCE Channel WITH Data <- Message, chan <- out

Init == /
   InChan!Init
   /
   OutChan!Init
   /
   q = << >>

TypeInvariant == /
   InChan!TypeInvariant
A.3. A CACHING MEMORY

\[
\begin{align*}
SSend(msg) &= \{\text{InChan!Send}(msg) \land \text{UNCHANGED } \langle\text{out, q}\rangle\} \\
BufRcv &= \{\text{InChan!Rcv} \land q' = \text{Append}(q, \text{in.val}) \land \text{UNCHANGED out}\} \\
BufSend &= \{q \# \langle\rangle \rightarrow \text{OutChan!Send}(\text{Head}(q)) \land q' = \text{Tail}(q) \land \text{UNCHANGED in}\} \\
RRcv &= \{\text{OutChan!Rcv} \land \text{UNCHANGED } \langle\text{in, q}\rangle\} \\
Next &= \{\forall \text{msg} \in \text{Message} : \text{SSend}(\text{msg}) \lor \text{BufRcv} \lor \text{BufSend} \lor \text{RRcv}\} \\
Spec &= \text{Init} \land \{\text{Next}\}_<<\text{in, out, q}>> \\
\text{THEOREM } Spec \Rightarrow \{\text{TypeInvariant}\}
\end{align*}
\]

--- MODULE FIFO ---

\[
\begin{align*}
\text{CONSTANT Message} \\
\text{VARIABLES in, out} \\
\text{Inner}(q) &= \text{INSTANCE InnerFIFO} \\
\text{Spec} &= \{\text{EE q : Inner}(q)\}_!\text{Spec}
\end{align*}
\]

--- MODULE MemoryInterface ---

\[
\begin{align*}
\text{VARIABLE memInt} \\
\text{CONSTANTS } \text{Send(\_\_, \_\_, \_\_, \_\_),} \\
&\quad \text{Reply(\_\_, \_\_, \_\_, \_\_),}
\end{align*}
\]

A.3. A Caching Memory
ASSUME \(\forall p, d, \text{miOld}, \text{miNew} : \)
\(\land \text{Send}(p,d,\text{miOld},\text{miNew}) \in \text{BOOLEAN} \)
\(\land \text{Reply}(p,d,\text{miOld},\text{miNew}) \in \text{BOOLEAN} \)
--------------------------------------------------------------
\(\text{MReq} == \{ \text{op} : \{"Rd"\}, \text{adr} : \text{Adr}\} \)
\(\cup \{ \text{op} : \{"Wr"\}, \text{adr} : \text{Adr}, \text{val} : \text{Val}\} \)
\(\text{NoVal} == \text{CHOOSE v : v \notin \text{Val}} \)
--------------------------------------------------------------

--- MODULE InternalMemory -----------------------------
EXTENDS MemoryInterface
VARIABLES mem, ctl, buf
--------------------------------------------------------------
\(\text{IInit} == \land \text{mem} \in \{\text{Adr} \rightarrow \text{Val}\} \)
\(\land \text{ctl} = \{p \in \text{Proc} | \rightarrow \"rdy"\} \)
\(\land \text{buf} = \{p \in \text{Proc} | \rightarrow \text{NoVal}\} \)
\(\land \text{memInt} \in \text{InitMemInt} \)

\(\text{TypeInvariant} == \)
\(\land \text{mem} \in \{\text{Adr} \rightarrow \text{Val}\} \)
\(\land \text{ctl} \in \{\text{Proc} \rightarrow \{\"rdy\", \"busy\", \"done\"\}\} \)
\(\land \text{buf} \in \{\text{Proc} \rightarrow \text{MReq} \cup \text{Val} \cup \{\text{NoVal}\}\} \)
A.3. A CACHING MEMORY

\[\text{Req}(p) == /\ \text{ctl}[p]\ = \text{"rdy"} \\
/\ \exists \text{req} \in \text{MReq} : \\
/\ \text{Send}(p, \text{req}, \text{memInt}, \text{memInt}') \\
/\ \text{buf}' = [\text{buf} \text{EXCEPT ![p] = req}] \\
/\ \text{ctl}' = [\text{ctl} \text{EXCEPT ![p] = "busy"}] \\
/\ \text{UNCHANGED mem}\]

\[\text{Do}(p) == \\
/\ \text{ctl}[p]\ = \text{"busy"} \\
/\ \text{mem}' = \text{IF} \ \text{buf}[p].\text{op} = \text{"Wr"} \\
\text{THEN} [\text{mem} \text{EXCEPT ![buf[p].adr] = buf[p].val}] \\
\text{ELSE mem} \\
/\ \text{buf}' = [\text{buf} \text{EXCEPT ![p] = IF buf[p].op = \text{"Wr"} \\
\text{THEN} \text{NoVal} \\
\text{ELSE mem[buf[p].adr]}] \\
/\ \text{ctl}' = [\text{ctl} \text{EXCEPT ![p] = "done"}] \\
/\ \text{UNCHANGED memInt}\]

\[\text{Rsp}(p) == /\ \text{ctl}[p]\ = \text{"done"} \\
/\ \text{Reply}(p, \text{buf}[p], \text{memInt}, \text{memInt}') \\
/\ \text{ctl}' = [\text{ctl} \text{EXCEPT ![p] = \text{"rdy"}]} \\
/\ \text{UNCHANGED \langle\langle mem, buf\rangle\rangle}\]

\[\text{INext} == \exists p \in \text{Proc}: \text{Req}(p) \lor \text{Do}(p) \lor \text{Rsp}(p)\]

\[\text{ISpec} == \text{IInit} /\ []\text{INext},<\text{memInt, mem, ctl, buf}>\]

\[\text{THEOREM ISpec} => []\text{TypeInvariant}\]

---------------------------------------- MODULE Memory ----------------------------------------
EXTENDS MemoryInterface
Inner(mem, ctl, buf) == INSTANCE InternalMemory
Spec == \EE mem, ctl, buf : Inner(mem, ctl, buf)!ISpec

----------------------------------------
-- MODULE WriteThroughCache --

EXTENDS Naturals, Sequences, MemoryInterface

VARIABLES mem, ctl, buf, cache, memQ

CONSTANT QLen

ASSUME (QLen \in Nat) \land (QLen > 0)

M == INSTANCE InternalMemory

Init == /
  M!IInit
  /
  cache = \{ p \in Proc |-> \{ a \in Adr |-> NoVal \} \}
  /
  memQ = << >>

TypeInvariant ==
  /
  mem \in \{ Adr \rightarrow Val \}
  /
  ctl \in \{ Proc \rightarrow \{ "rdy", "busy", "waiting", "done" \} \}
  /
  buf \in \{ Proc \rightarrow MReq \cup Val \cup \{ NoVal \} \}
  /
  cache \in \{ Proc \rightarrow \{ Adr \rightarrow Val \cup \{ NoVal \} \} \}
  /
  memQ \in Seq(Proc \times MReq)

Coherence == \forall p, q \in Proc, a \in Adr :
  (NoVal \notin \{ cache[p][a], cache[q][a] \})
  \rightarrow (cache[p][a] = cache[q][a])

Req(p) == M!Req(p) \land UNCHANGED <<cache, memQ>>

Rsp(p) == M!Rsp(p) \land UNCHANGED <<cache, memQ>>

RdMiss(p) ==
  /
  (ctl[p] = "busy") \land (buf[p].op = "Rd")
  /
  cache[p][buf[p].adr] \# NoVal
  /
  Len(memQ) < QLen
  /
  memQ' = Append(memQ, <<p, buf[p]>>)
  /
  ctl' = [ctl EXCEPT ![p] = "waiting"]
  /
  UNCHANGED <<memInt, mem, cache, memQ>>

DoRd(p) ==
  /
  ctl[p] \in \{ "busy", "waiting" \}
  /
  buf[p].op = "Rd"
  /
  cache[p][buf[p].adr] \# NoVal
  /
  buf' = [buf EXCEPT ![p] = cache[p][buf[p].adr]]
  /
  ctl' = [ctl EXCEPT ![p] = "done"]
  /
  UNCHANGED <<memInt, mem, cache, memQ>>

DoWr(p) ==
  LET r == buf[p]
  IN
  (ctl[p] = "busy") \land (r.op = "Wr")
A.3. A CACHING MEMORY

\[\text{Len}(\text{memQ}) \leq \text{QLen}\]
\[\text{cache'} = \{q \in \text{Proc} | \rightarrow\]
\[\quad \text{IF (p}=q) \quad \text{OR} \quad \text{cache}[q][r.adr] \neq \text{NoVal}\]
\[\quad \text{THEN [cache}[q] \text{ EXCEPT } !\{r.adr\} = r.val\]
\[\quad \text{ELSE cache}[q] \}\]
\[\text{memQ'} = \text{Append}(\text{memQ}, \langle\langle p, r\rangle\rangle)\]
\[\text{buf'} = [\text{buf EXCEPT } !\{p\} = \text{NoVal}]\]
\[\text{ctl'} = [\text{ctl EXCEPT } !\{p\} = "done"]\]
\[\text{UNCHANGED } \langle\langle \text{memInt}, \text{mem} \rangle\rangle\]

\[\text{vmem} =\]
\[\quad \text{LET } f[i \text{ in } 0 .. \text{Len}(\text{memQ})] =\]
\[\quad \quad \text{IF } i=0 \text{ THEN mem}\]
\[\quad \quad \text{ELSE IF memQ[i][2].op = "Rd"}\]
\[\quad \quad \quad \text{THEN } f[i-1]\]
\[\quad \quad \quad \text{ELSE } [f[i-1] \text{ EXCEPT } !\{memQ[i][2].adr\} = memQ[i][2].val]\]
\[\quad \text{IN } f[\text{Len}(\text{memQ})]\]

\[\text{MemQWr} = \text{LET } r = \text{Head}(\text{memQ})[2]\]
\[\quad \text{IN } \langle\langle \text{memQ} \# \langle\rangle \rangle \text{ OR } (r\text{.op = "Wr"})\]
\[\quad \langle\langle \text{mem'} = [\text{mem EXCEPT } !\{r\.adr\} = r.val]\]
\[\quad \quad \langle\langle \text{memQ'} = \text{Tail}(\text{memQ})\]
\[\quad \quad \text{UNCHANGED } \langle\langle \text{memInt}, \text{mem}, \text{buf}, \text{ctl}, \text{cache}\rangle\rangle\]

\[\text{MemQRd} =\]
\[\quad \text{LET } p = \text{Head}(\text{memQ})[1]\]
\[\quad r = \text{Head}(\text{memQ})[2]\]
\[\quad \langle\langle \text{memQ} \# \langle\rangle \rangle \text{ OR } (r\text{.op = "Rd"})\]
\[\quad \langle\langle \text{memQ'} = \text{Tail}(\text{memQ})\]
\[\quad \langle\langle \text{cache'} = [\text{cache EXCEPT } !\{p\.adr\} = \text{vmem}[r\.adr]\]
\[\quad \quad \text{UNCHANGED } \langle\langle \text{memInt}, \text{mem}, \text{buf}, \text{ctl}\rangle\rangle\]

\[\text{Evict}(p, a) =\]
\[\quad \langle\langle \text{ctl}[p] = "waiting" \Rightarrow (\text{buf}[p].adr \neq a)\]
\[\quad \langle\langle \text{cache'} = [\text{cache EXCEPT } !\{p\.a\} = \text{NoVal}\]
\[\quad \quad \text{UNCHANGED } \langle\langle \text{memInt}, \text{mem}, \text{buf}, \text{ctl}, \text{memQ}\rangle\rangle\]

\[\text{Next} =\]
\[\quad \langle\langle \text{E p in Proc : \langle\langle Req(p) OR Rsp(p)\rangle\rangle \text{ OR RdMiss(p) OR DoRd(p) OR DoWr(p) OR E a in Adr : Evict(p, a)\rangle\rangle}\]
\[\quad \langle\langle \text{MemQWr OR MemQRd}\rangle\rangle\]

\[\text{Spec} =\]
APPENDIX A. THE ASCII SPECIFICATIONS

Init /
\[] [Next].<<memInt, mem, buf, ctl, cache, memQ>>

--------------------------------------------------------------
THEOREM Spec => [] (TypeInvariant /\ Coherence)
--------------------------------------------------------------
LM == INSTANCE Memory
THEOREM Spec => LM!Spec

================================================================================================

A.4 The Alternating Bit Protocol

------------------- MODULE AlternatingBit -------------------
EXTENDS Naturals, Sequences
CONSTANTS Data
VARIABLES msgQ, ackQ, sBit, sAck, rBit, sent, rcvd

ABInit == /
\ msgQ = << >>
\ ackQ = << >>
\ sBit \in \{0, 1\}
\ sAck = sBit
\ rBit = sBit
\ sent = << >>
\ rcvd = << >>

TypeInv == /
\ msgQ \in Seq(0,1) \X Data)
\ ackQ \in Seq(0,1))
\ sBit \in \{0, 1\}
\ sAck \in \{0, 1\}
\ rBit \in \{0, 1\}
\ sent \in Seq(Data)
\ rcvd \in Seq(Data)

SndNewValue(d) == /
\ sAck = sBit
\ sent’ = Append(sent, d)
\ sBit’ = 1 - sBit
\ msgQ’ = Append(msgQ, <<sBit’, d>>) 
\ UNCHANGED <<ackQ, sAck, rBit, rcvd>>

ReSndMsg ==
\ sAck # sBit
\ msgQ’ = Append(msgQ, <<sBit, sent[Len(sent)]>>) 
\ UNCHANGED <<ackQ, sBit, sAck, rBit, sent, rcvd>>
A.4. THE ALTERNATING BIT PROTOCOL

RcvMsg ==
\/
msgQ' = Tail(msgQ)
\/
rBit' = Head(msgQ)[1]
\/
rcvd' = IF rBit' # rBit THEN Append(rcvd, Head(msgQ)[2])
ELSE rcvd
\/
UNCHANGED <<ackQ, sBit, sAck, sent>>

SndAck ==
\/
ackQ' = Append(ackQ, rBit)
\/
UNCHANGED <<msgQ, sBit, sAck, rBit, sent, rcvd>>

RcvAck ==
\/
ackQ' = Tail(ackQ)
\/
sAck' = Head(ackQ)
\/
UNCHANGED <<msgQ, sBit, rBit, sent, rcvd>>

Lose(c) ==
\/
c' = \[j \in 1..(Len(c)-1) |-> IF j \leq i THEN c[j]
ELSE c[j+1] \]
\/
UNCHANGED <<sBit, sAck, rBit, sent, rcvd>>

LoseMsg == Lose(msgQ) /
UNCHANGED ackQ

LoseAck == Lose(ackQ) /
UNCHANGED msgQ

ABNext ==
\/
E d \in Data : SndNewValue(d)
\/
ReSndMsg \/
RcvMsg \/
SndAck \/
RcvAck
\/
LoseMsg \/
LoseAck

vars == << msgQ, ackQ, sBit, sAck, rBit, sent, rcvd>>

Spec == ABInit /
[|[ABNext]|]_vars

Inv ==
\/
Len(rcvd) \in \{Len(sent)-1, Len(sent)\}
\/
A i \in 1..Len(rcvd) : rcvd[i] = sent[i]

THEOREM Spec => []Inv

-------------------------------------------------------------

------------------ MODULE MCAlternatingBit ------------------
EXTENDS AlternatingBit
CONSTANTS msgQLen, ackQLen, sentLen

ASSUME /
\ msgQLen \in Nat
\ ackQLen \in Nat
\ sentLen \in Nat

SeqConstraint == /
\ Len(msgQ) \leq msgQLen
\ Len(ackQ) \leq ackQLen
\ Len(sent) \leq sentLen

====================================================================
Bibliography


