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 Specify

Writing a specification requires effort; the benefit it provides must justify that effort. The purpose of writing a specification is to help avoid errors. Here are some ways it can do that.

- 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. The most useful tool written so far for this purpose is the TLC model checker, described in Chapter 14.

Whether the benefit justifies the effort of writing the specification depends on the nature of the project. Specification is not an end in itself; it is just a tool that an engineer should be able to use when appropriate.

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 model. 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.

The primary 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, when writing a TLA$^+$ specification, you will probably concentrate your efforts on the parts of the system that are most likely to have such errors. If that's not where you should be concentrating your efforts, then you probably shouldn't be using TLA$^+$.

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 granularity that can help. To describe it, we

need the TLA$^+$ action-composition operator "$\cdot$". If A and B are actions, then the action $A \cdot B$ is executed by executing first A then B as a single step. More precisely, $A \cdot B$ is the action defined by letting $s \to t$ be an $A \cdot B$ step iff there exists a state u such that $s \to u$ is an A step and $u \to t$ 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 $R \vee L$. When its execution is represented by a single step, the operation is described with the action $R \cdot L$.[1] Let $S2$ be the finer-grained specification in which the operation is executed in two steps, and let $S1$ 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 $S1$ or $S2$ as the specification. Let's examine the relation between the two specifications.

We can transform any behavior σ satisfying $S1$ into a behavior $\widehat{\sigma}$ satisfying $S2$ by replacing each step $s \overset{R \cdot L}{\to} t$ with the pair of steps $s \overset{R}{\to} u \overset{L}{\to} t$, for some state u. If we regard σ as being equivalent to $\widehat{\sigma}$, then we can regard $S1$ as being a strengthened version of $S2$—one that allows fewer behaviors. Specification $S1$ requires that each R step be followed immediately by an L step, while $S2$ 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 $S2$ are important.

The additional behaviors allowed by $S2$ are not important if the actual system executions they describe are also described by behaviors allowed by $S1$. So, we can ask whether each behavior τ satisfying $S2$ has a corresponding behavior $\widetilde{\tau}$ satisfying $S1$ that is, in some sense, equivalent to τ. One way to construct $\widetilde{\tau}$ from τ is to transform a sequence of steps

$$s \overset{R}{\to} u_1 \overset{A_1}{\to} u_2 \overset{A_2}{\to} u_3 \ \ldots \ u_n \overset{A_n}{\to} u_{n+1} \overset{L}{\to} t \tag{7.1}$$

into the sequence

$$s \overset{A_1}{\to} v_1 \ \ldots \ v_{k-2} \overset{A_k}{\to} v_{k-1} \overset{R}{\to} v_k \overset{L}{\to} v_{k+1} \overset{A_{k+1}}{\to} v_{k+2} \ \ldots \ v_{n+1} \overset{A_n}{\to} t \tag{7.2}$$

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.

[1] We 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.

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 (i) each one leaves unchanged any variable whose value may be changed by the other, and (ii) 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:

- 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 $O_1 \vee O_2 \vee \ldots \vee O_m$ 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, ..., 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 $s \stackrel{O_1}{\rightarrow} \cdots \stackrel{O_m}{\rightarrow} t$ 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 program 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 program 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 returning 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, which 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 need and will 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 8. 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 probably 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 $q = \langle h' \rangle \circ q'$ may look like a nice, short way of writing

(7.3) $(h' = \mathit{Head}(q)) \land (q' = \mathit{Tail}(q))$

But not only is $q = \langle h' \rangle \circ q'$ 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 $h' = \mathit{Head}(q)$ and $q' = \mathit{Tail}(q)$ are the only values of h' and q' that satisfy $q = \langle h' \rangle \circ q'$. There could be other values of h' and q', which are not sequences, that satisfy the formula.

In general, the best way to specify the new value of a variable v is with a conjunct of the form $v' = \mathit{exp}$ or $v' \in \mathit{exp}$, where exp is a state function—an expression with no primes.

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 Nat. You may be tempted then to 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 even if $n' =$ "abc". The meaning of the formula is not changed just because you've defined a type invariant that asserts $n \in \mathit{Nat}$.

In general, you may want to describe the new value of a variable x by asserting some property of x'. However, the next-state action should imply that x' is an element of some suitable set. For example, a specification might define[2]

$$\begin{aligned} \mathit{Action}1 &\triangleq (n' > 7) \land \ldots \\ \mathit{Action}2 &\triangleq (n' \leq 6) \land \ldots \\ \mathit{Next} &\triangleq (n' \in \mathit{Nat}) \land (\mathit{Action}1 \lor \mathit{Action}2) \end{aligned}$$

[2] An alternative approach is to define Next to equal $\mathit{Action}1 \lor \mathit{Action}2$ and to let the specification be $\mathit{Init} \land \Box[\mathit{Next}]_{\ldots} \land \Box(n \in \mathit{Nat})$. But it's usually better to stick to the simple form $\mathit{Init} \land \Box[\mathit{Next}]_{\ldots}$ for specifications.

Don't be too abstract.

Suppose a user interacts with the system by typing on a keyboard. We could describe the interaction abstractly with a variable *typ* and an operator parameter *KeyStroke*, where the action *KeyStroke*("a", *typ*, *typ*′) represents the user typing an "a". This is the approach we took in describing the communication between the processors and the memory in the *MemoryInterface* module on page 48.

A more concrete description would be to let *kbd* represent the state of the keyboard, perhaps letting $kbd = \{\}$ mean that no key is depressed, and $kbd = \{\text{"a"}\}$ mean that the *a* key is depressed. The typing of an *a* is represented by two steps, a $[kbd = \{\}] \to [kbd = \{\text{"a"}\}]$ step represents the pressing of the *a* key, and a $[kbd = \{\text{"a"}\}] \to [kbd = \{\}]$ step represents its release. This is the approach we took in the asynchronous interface specifications of Chapter 3.

The abstract interface is simpler; typing an *a* is represented by a single *KeyStroke*("a", *typ*, *typ*′) step instead of a pair of steps. However, using the concrete representation leads us naturally to ask: what if the user presses the *a* key and, before releasing it, presses the *b* key? That's easy to describe with the concrete representation. The state with both keys depressed is $kbd = \{\text{"a"}, \text{"b"}\}$. Pressing and releasing a key are represented simply by the two actions

$$Press(k) \triangleq kbd' = kbd \cup \{k\} \qquad Release(k) \triangleq kbd' = kbd \setminus \{k\}$$

The possibility of having two keys depressed cannot be expressed with the simple abstract interface. To express it abstractly, we would have to replace the parameter *KeyStroke* with two parameters *PressKey* and *ReleaseKey*, and we would have to express explicitly the property that a key can't be released until it has been depressed, and vice-versa. The more concrete representation is then simpler.

We might decide that we don't want to consider the possibility of two keys being depressed, and that we prefer the abstract representation. But that should be a conscious decision. Our abstraction should not blind us to what can happen in the actual system. When in doubt, it's safer to use a concrete representation that more accurately describes the real system. That way, you are less likely to overlook real problems.

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,

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

We know that these two values are different because they have different *type* fields.

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

$$\begin{array}{lcl} Up & \triangleq & \exists\, e \in Elevator : \ldots \\ Down & \triangleq & \exists\, e \in Elevator : \ldots \\ Move & \triangleq & Up \lor Down \end{array}$$

it's usually better to write

$$\begin{array}{lcl} Up(e) & \triangleq & \ldots \\ Down(e) & \triangleq & \ldots \\ Move & \triangleq & \exists\, e \in Elevator : Up(e) \lor Down(e) \end{array}$$

Prime only what you mean to prime.

When writing an action, be careful where you put your primes. The expression $f[e]'$ equals $f'[e']$; it equals $f'[e]$ only if $e' = e$, which need not be true if the expression e contains variables. Be especially careful when priming an operator whose definition contains a variable. For example, suppose x is a variable and op is defined by

$$op(a) \triangleq x + a$$

Then $op(y)'$ equals $(x+y)'$, which equals $x'+y'$, while $op(y')$ equals $x+y'$. There is no way to use op and $'$ to write the expression $x' + y$. (Writing $op'(y)$ doesn't work because it's illegal—you can prime only an expression, not an operator.)

Write comments as comments.

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

$$\begin{array}{lcl} A & \triangleq & \lor \land\ x \geq 0 \\ & & \quad \land \ldots \\ & & \lor \land\ x < 0 \\ & & \quad \land \text{FALSE} \end{array}$$

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 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

$$\begin{array}{lcl} A & \triangleq & \land\ x \geq 0 \quad A \text{ is not enabled if } x < 0 \\ & & \land \ldots \end{array}$$

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 most 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

$$
\begin{array}{l}
RdMiss(p) \triangleq \text{ Enqueue a request to write value from memory to } p\text{'s cache.} \\
\quad \text{Some enabling condition must be conjoined here.} \\
\quad \land\ memQ' = Append(memQ, buf[p]) \qquad \text{Append request to } memQ. \\
\quad \land\ ctl' = [ctl \text{ EXCEPT } ![p] = \text{“?”}] \qquad \text{Set } ctl[p] \text{ to value to be determined later.} \\
\quad \land \text{ UNCHANGED } \langle memInt,\ wmem,\ buf,\ cache \rangle
\end{array}
$$

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.