Chapter 1

Describing Systems

The fundamental scientific achievement of the past 2500 years has been the use of mathematics to describe the physical universe. In this Chapter, we explain how to use mathematics to describe discrete systems. Our explanation is based on three very simple systems, culminating in one that we call the \textit{alternation} system. This simple system provides an abstract view of an important class of real systems with which we will begin our study of concurrency. We also introduce the TLA* language for writing mathematical descriptions of discrete systems, and the TLC program for checking these descriptions.

1.1 A Discrete Oscillator

In science, a continuous system is described by explaining how its state evolves with time. For example, a one-dimensional oscillator might be described by giving the value of a variable $x$ as a function of time $t$:

\[ x(t) = 1 - \sin(2\pi t). \]
An analogous component in a computer system is described by a function like this:

\[ x \]

\[ t \]

Such a component approximates a discrete oscillator whose state variable \( x \) assumes only the two possible values 0 and 1:

\[ x \]

\[ t \]

In computer science, we are usually interested only in the sequence of states assumed by a discrete system, not in the amount of time spent in each state. We don’t want to distinguish a discrete oscillator that behaves as shown above from one that behaves like this:

\[ x \]

\[ t \]

Instead of describing the state of a system as a function of time, we describe it as a sequence of values. For the discrete oscillator, this sequence is:

\[ (1.1) \quad [x = 0] \rightarrow [x = 1] \rightarrow [x = 0] \rightarrow [x = 1] \rightarrow \ldots \]

The arrows are just decoration, meant to suggest that each state changes directly into the next one. We call such a sequence of states a behavior.

We want to describe a system’s behavior by a formula, not by listing the complete sequence of states. We can describe the behavior of our discrete oscillator by saying that the \( i^{th} \) value of \( x \) is 1 if \( i \) is even and 0 if \( i \) is odd. We can express this with a formula using the operator \( \% \), which defined so that \( i \% n \) is...
1.1. A DISCRETE OSCILLATOR

the “remainder when i is divided by n”. The \( i^{\text{th}} \) value of \( x \) can then be written \((i + 1) \mod 2\). However, systems are seldom simple enough for us to be able to express the \( i^{\text{th}} \) state explicitly as a function of \( i \). We need a different approach.

Most continuous systems are also too complicated for their state to be described as an explicit function of time. Such systems are described by giving (a) the initial state and (b) a differential equation describing how the state changes at any time. We use an analogous approach for describing a discrete system: (a) we give the initial state and (b) for each point in the behavior, we give the next state as a function of the current state. For the discrete oscillator:

(a) The initial state is given by \( x = 0 \).

(b) At any point in the behavior, if \( x \) denotes the variable’s value in the current state and \( x' \) denotes its value in the next state, then \( x' = (x + 1) \mod 2 \).

We call the formula \( x = 0 \) the initial predicate and the formula \( x' = (x + 1) \mod 2 \) the next-state action of the discrete oscillator.

We combine the initial predicate and next-state action into this single formula:

\[
\begin{align*}
(x = 0) & \land (x' = (x + 1) \mod 2) \\
\text{Gives the value of} & \\
\text{x in the initial state.} & \text{Gives the new value} x' \text{ of x as} \text{a function of its current value.} & \text{At every point in the behavior:}
\end{align*}
\]

In this formula, \( x \) means the current value of \( x \), and \( x' \) means the value of \( x \) in the next state. The entire formula is a statement about the beginning of the behavior, and \( \Box(x) \) means that \( (\ldots) \) is true at all points in the behavior. The first conjunct therefore asserts that the current value of \( x \), at the beginning of the behavior, is 0. The second conjunct asserts that, at all points in the behavior \( (\Box) \), the value \( x_{\text{new}} \) of \( x \) in the next state is related to its value \( x_{\text{old}} \) in the current state by \( x_{\text{new}} = (x_{\text{old}} + 1) \mod 2 \).

In describing the discrete oscillator mathematically, we have used the following kinds of formulas:

- A formula like \( (x + 1) \mod 2 \) that contains only unprimed variables (and no \( \Box \)) is called a state function. A Boolean-valued state function, like \( x = 0 \), is called a state predicate. A state predicate is an assertion about (is true or false for) an individual state.

- A formula like \( x' = (x + 1) \mod 2 \) that may contain primed and unprimed variables, but no \( \Box \), is called an action. An action is an assertion about a pair of states—an old state described by the unprimed variables and a new state described by a primed variable.
state described by the primed variables. A state predicate is a degenerate action that contains no primed variables and is thus an assertion only about the old state.

- A formula like \( (x = 0) \land \Box(x' = (x + 1) \mod 2) \) that contains a \( \Box \) is called a temporal formula. A temporal formula is an assertion about a behavior, which is a sequence of states. An action is a degenerate temporal formula that is an assertion only about the first two states of a behavior. (A state function is thus a degenerate temporal formula that is an assertion only about the first state of a behavior.)

We have called \( (x = 0) \land \Box(x' = (x + 1) \mod 2) \) a description of the discrete oscillator. We consider it a specification of the oscillator. A system’s specification defines what it means for a sequence of states to be a (correct) behavior of the system.

## 1.2 Alternation

### 1.2.1 The Basic Specification

A concurrent system can exhibit many different types of interactions among its components. The simplest and most common is alternation, in which the system repeatedly executes two operations. The discrete oscillator provides one example of alternation, in which the two operations are setting \( x \) to 0 and to 1.

We describe alternation in terms of two operations Call and Return. Here are some examples of this kind of interaction:

- **Call** represents calling a procedure in a program, and **Return** represents the return from the procedure.

- **Call** represents sending a character to a printer, **Return** represents an acknowledgement that the character has been printed.

- **Call** represents issuing a read or write operation to a disk, **Return** represents the response—either the data on a read or an acknowledgement on a write.

For the moment, we won’t worry about what the operations Call and Return do. We assume that these operations are described by two actions, named Call and Return, which we’ll define later.

To describe this system, we introduce a variable whose value indicates which operation is to be performed next. We will use a variable \( x \) that describes a discrete oscillator, so \( x = 0 \) means that the next operation to be performed is Call, and \( x = 1 \) means that the next operation to be performed is Return.
1.2. ALTERNATION

We describe the system’s behaviors with the formula,

$$\text{AltSpec} \triangleq \text{AltInit} \land \Box \text{AltNext}$$

We have to define the initial predicate \text{AltInit} and the next-state action \text{AltNext}. To define \text{AltInit}, we let \text{CRInit} be the formula describing the initial state of whatever variables are mentioned by \text{Call} and \text{Return}. Then we have

$$\text{AltInit} \triangleq \text{CRInit} \land (x = 0)$$

There are two kinds of steps (state changes) that occur: ones that perform the \text{Call} step and ones that perform \text{Return} steps. A step of the system is either one or the other. “Or” is written in mathematics as $\lor$, so the next-state action, which we’ll call \text{AltNext}, is defined by:

$$\text{AltNext} \triangleq \text{AltCall} \lor \text{AltReturn}$$

where \text{AltCall} performs the \text{Call} step and \text{AltReturn} performs the \text{Return} step. An \text{AltCall} step can be performed when $x$ equals 0. It performs the \text{Call} step and sets $x$ to 1. Its definition is

$$\text{AltCall} \triangleq (x = 0)$$

\begin{align*}
\land \text{Call} \\
\land (x' = 1)
\end{align*}

The definition of \text{AltReturn} is analogous:

$$\text{AltReturn} \triangleq (x = 1)$$

\begin{align*}
\land \text{Return} \\
\land (x' = 0)
\end{align*}

We’re going to be writing a lot of formulas with conjunctions and disjunctions. To make them easier to read, we will usually write them as lists bulleted with $\land$ and $\lor$, using indentation to eliminate parentheses. So, let’s rewrite these definitions as:

$$\text{AltCall} \triangleq \land x = 0 \land \text{Call} \land x' = 1$$
$$\text{AltReturn} \triangleq \land x = 1 \land \text{Return} \land x' = 0$$

1.2.2 Call and Return

We now define the \text{Call} and \text{Return} actions used in our specification of the \text{Alternation} system. These actions represent the operations of issuing a call or a return with a value. The way the call and return operations are implemented will depend on the particular type of system. If they represent software procedure calls, then they are implemented by manipulating a call stack. If they represent
interaction between hardware components, then they might be implemented by setting voltage levels on wires—*Call* setting the levels on one set of wires and *Return* setting the levels on a different set of wires. We will define the actions in a way that resembles this hardware implementation. We introduce variables *arg* and *rtn*, where the value of *arg* represents the levels on the wires set by the *Call* action, and the value of *rtn* represents the levels on the wires set by *Return*. More precisely, the values of *arg* and *rtn* are the values represented by the voltages on those two sets of wires.

Let *Input* be the set of possible call values. A *Call* step can set *arg* to any element of *Input*, and it leaves *rtn* unchanged. In other words, a step is a *Call* step iff the value of *arg* in the new state is an element of *Input* and the value of *rtn* in the new state equals its value in the old state. So, we define:

\[
\text{Call} \triangleq \land \arg' \in \text{Input} \\
\land \rtn' = \rtn
\]

A *Return* step sets *rtn* to the return value and leaves *arg* unchanged. In classical computing paradigms, the return value is a function of the call value. We let *Rtn Val(v)* be the return value for a call value *v*. The definition of *Return* is then

\[
\text{Return} \triangleq \land \rtn' = \text{Rtn Val}(\arg) \\
\land \arg' = \arg
\]

We won’t worry now about the definition of the initial predicate *CRInit*.

### 1.3 Alternation in TLA⁺

So far, we have been writing our specifications in ordinary mathematics, without worrying about the precise rules for writing mathematics. We now introduce a precise language, called TLA⁺, for writing specifications. Writing in a precise language, with a well-defined semantics, allows us to be sure that we know exactly what a specification means. But, more important, writing in a precise language allows us to use a computer to check our specifications. We will be able to check that our specifications meet some relatively simple conditions (like being syntactically correct and being “type safe”) as well as more interesting conditions.

#### 1.3.1 The Alternation Module

We now write a TLA⁺ specification of the alternation system. In TLA⁺, specifications are organized into modules. The definition of our specification *AltSpec* of the two-operation system is in a module that we call *Alternation*, which appears in Figure 1.1 on page 8. The module begins:
1.3. ALTERNATION IN TLA⁺

---

**MODULE Alternation**

We will put the definitions of *Call*, *Return*, and *CRInit* in another module named *CallReturn*. The next statement imports the definitions from that module into the current module.

`EXTENDS CallReturn`

The module next declares the variable *x* with the statement:

`VAR x`

Next comes a horizontal bar that is purely decorative. It is followed by the definitions of *AltInit*, *AltCall*, etc. more or less the way they appear above. A significant difference is that, because identifiers must be declared or defined before they are used, the definition of *AltReturn* has to follow the definitions of *AltCal* and *AltReturn*.

After another decorative horizontal bar, we come to the definition of formula *AltSpec*, which is the specification of the system. Instead of defining *AltSpec* to equal *AltInit* ∧ □*AltNext*, the module defines it to be the strange formula

\[ *AltInit* \land \square[*AltNext*]|_{\text{iFace, } x} \]

The mysterious \(|\_\text{iFace, } x\rangle\) surrounding *AltNext* is explained later in these notes. We ignore it for now and pretend that *AltSpec* is defined by

\[ *AltSpec* \triangleq \cdot *AltInit* \land \square *AltNext* \]

The module ends with

---

Figure 1.1 is the typeset version of the module. Figure 1.2 on the next page shows how you type the module as an ordinary ASCII file. TLA⁺ files use the extension tla, so module *Alternation* should be in file *Alternation.tla*. Most of what you type is obvious. There are special sequences for typing some symbols—for example, \(\triangleq\) is typed \(\approx\), and \(\land\) is typed \(\\backslash\). From now on, we will show only the typeset version of a specification. Table A.2 on page 168 describes how to type all the TLA⁺ symbols.

### 1.3.2 The *CallReturn* Module

We now write the *CallReturn* module. The module begins with:

---

**MODULE CallReturn**

The module must describe the set Input of input values and the operator *RtnVal*. The natural way to do this would be to let them be unspecified parameters, which we would do by writing
MODULE Alternation

EXTENDS CallReturn

VARIABLE x

\[\text{AltInit} \triangleq CRInit \land (x = 0)\]

\[\text{AltCall} \triangleq \land x = 0 \land Call \land x' = 1\]

\[\text{AltReturn} \triangleq \land x = 1 \land Return \land x' = 0\]

\[\text{AltNext} \triangleq \text{AltCall} \lor \text{AltReturn}\]

\[\text{AltSpec} \triangleq \text{AltInit} \land [\text{AltNext}]_{iFace, x}\]

Figure 1.1: The TLA module for the Alternation System

---

EXTENDS CallReturn

VARIABLE x

--

\[\text{AltInit} \equiv CRInit \land (x = 0)\]

\[\text{AltCall} \equiv \land x = 0 \land Call \land x' = 1\]

\[\text{AltReturn} \equiv \land x = 1 \land Return \land x' = 0\]

\[\text{AltNext} \equiv \text{AltCall} \lor \text{AltReturn}\]

--

\[\text{AltSpec} \equiv \text{AltInit} \land [\text{AltNext}]_{iFace, x}\]

---

Figure 1.2: How to type the Alternation module.
1.4. INVARIANCE

CONSTANTS Input, RtnVal(\texttt{\_})

We would then give values to these parameters when using TLC to check the specification. However, just to have a definite example, we define Input to be the set \{0,1,\ldots,15\} of “four-bit numbers” and define \texttt{RtnVal(v)} to equal \texttt{v^2}. For this, we need an exponentiation (or multiplication) operator. The arithmetic operators are not built-in TLA⁺ operators. Instead, they are defined in standard modules that can be imported. The usual arithmetic operators on natural numbers are defined in the standard Naturals module. This module also defines the operator “\ldots” so that \texttt{i..j} is the set of natural numbers \texttt{n} such that \texttt{i \leq n \leq j}. The module includes the definitions from the Naturals module with the statement

EXTENDS Naturals

It then defines \texttt{Input} and \texttt{RtnVal} by

\begin{align*}
\texttt{Input} & \triangleq 0..15 \\
\texttt{RtnVal(n)} & \triangleq n^2
\end{align*}

Note that the exponentiation operator is typed as \texttt{^} (caret), so \texttt{n^2} is typed as \texttt{\texttt{n^2}}.

The module next declares the variables \texttt{arg} and \texttt{rtn}, and defines the initial predicate \texttt{CRInit} to assert that the values of \texttt{arg} and \texttt{rtn} are both 0. The complete module, which appears in Figure 1.3 on the next page, is straightforward—except for the last definition

\texttt{iFace} \triangleq \langle \texttt{arg, rtn} \rangle

It defines the identifier \texttt{iFace} to equal the pair of values \langle \texttt{arg, rtn} \rangle. For example, in the initial state, \texttt{iFace} equals \langle 0,0 \rangle. This explains the meaning of the \texttt{iFace} that appears in the mysterious \texttt{\texttt{\mid \mid \{iFace, a\}}} in the definition of \texttt{AltSpec}. But, we are not yet ready to explain why we introduced \texttt{iFace} and what it’s doing there. So, you should continue to ignore it.

1.3.3 The Syntactic Analyzer

Now that we have a precise description of the specification, we can apply computerized tools to check it. The first thing we can do is run the TLA⁺ Syntactic Analyzer to make sure that there are no syntax errors. Section B.2 of the appendix describes how to do that.

1.4 Invariance

If you are used to programming languages, you may have been surprised to notice that the variable declarations in modules \texttt{Alternate} and \texttt{CallReturn} did
EXTENDS Naturals

\[
\begin{align*}
\text{Input} &\triangleq 0 \ldots 15 \\
\text{Rtn Val}(n) &\triangleq n^2
\end{align*}
\]

VARIABLES arg, rtn

\[
\begin{align*}
\text{CRInit} &\triangleq \land \text{arg} = 0 \\
&\land \text{rtn} = 0
\end{align*}
\]

Call \triangleq \land \text{arg}' \in \text{Input} \\
&\land \text{rtn}' = \text{rtn}

\[
\begin{align*}
\text{Return} &\triangleq \land \text{rtn}' = \text{Rtn Val}(\text{arg}) \\
&\land \text{arg}' = \text{arg}
\end{align*}
\]

iFace \triangleq (\text{arg}, \text{rtn})

Figure 1.3: The CallReturn module.

not specify the types of the variables. Types are used in programming languages, among other reasons, to catch programming errors early: if a variable is used in a manner inconsistent with its type, then the compiler can flag an error. The same can be done in TLA by through assertions about the values a variable can have. For example, we can prove that in any behavior satisfying AltSpec, the value of \( x \) is always 0 or 1. This follows by a simple induction argument from the observations that:

- \( \text{AltInit} \) asserts that \( x \) initially equals 0.
- \( \text{AltNext} \) asserts that any step leaves \( x \) equal to 0 or 1.

Another way to say that \( x \) always equals 0 or 1 is that the formula \( x \in \{0, 1\} \) is always true. Remembering that \( \Box \) means always, we can write this assertion \( \Box(x \in \{0, 1\}) \). Our observation is that \( \Box(x \in \{0, 1\}) \) is true for all behaviors satisfying AltSpec. Another way of saying this is that, for any behavior, if AltSpec is true of the behavior then \( \Box(x \in \{0, 1\}) \) is also true of the behavior. In other words, the formula

\[
\text{AltSpec} \Rightarrow \Box(x \in \{0, 1\})
\]

is true for all behaviors. A temporal formula that’s true for all behaviors is called a (temporal) theorem.

In general, if Spec is a temporal formula and Inv is a state predicate such that \( \text{Spec} \Rightarrow \Box\text{Inv} \) is a theorem, then we say that Inv is an invariant of Spec,
or of the system described by Spec. Thus, we say that \( x \in \{0, 1\} \) is an invariant of \( \text{AltSpec} \), or that it is an invariant of the alternation system. Invariants can express important properties of a system. An invariant of the form \( v \in S \), for \( v \) a variable, is called a \textit{type invariant}. When such an invariant holds, we say that \( v \) has \textit{type} \( S \) in the specification. Thus, \( x \) has type \( \{0, 1\} \) in the alternation specification.

It’s generally a good idea to write a type invariant for a specifications. Knowing the types of the variables helps the reader understand the specification. Let’s write a type invariant for the variables of \textit{CallReturn}. The variable \( \text{arg} \) has type \textit{Input}, which equals \( 0 \ldots 15 \). Since \( n \in 0 \ldots 15 \) implies \( n^2 \in 0 \ldots 255 \), it’s easy to see that \( \text{rtn} \) has type \( 0 \ldots 255 \).

For symmetry, let’s define \textit{Output} to equal \( 0 \ldots 255 \) in module \textit{CallReturn}. Then we can define the type invariant \( \text{CRInvariant} \) in module \textit{CallReturn} by

\[
\text{CRInvariant} \triangleq \text{arg} \in \text{Input} \\
\text{\hspace{2cm} \& \text{rtn} \in \text{Output}}
\]

Module \textit{CallReturn} doesn’t define a complete specification; it just defines the \textit{Call} and \textit{Return} actions. We expect \( \text{CRInvariant} \) to be an invariant of a specification that uses these actions. In our case, that specification is \textit{AltSpec} in \textit{Alternation}. So, we add to module \textit{Alternation} the definition

\[
\text{AltInvariant} \triangleq \text{CRInvariant} \land (x \in \{0, 1\})
\]

State predicate \( \text{AltInvariant} \) is the complete type invariant of the alternation specification, describing the types of all its variables. We also add to module \textit{Alternation} the statement

\[
\text{THEOREM} \quad \text{AltSpec} \Rightarrow \Box \text{AltInvariant}
\]

It asserts that \( \text{AltInvariant} \) is, indeed, an invariant of \( \text{AltSpec} \).

Invariants play a crucial role in understanding concurrent systems. The key to why an algorithm works lies in an invariant that it satisfies. Because our alternation specification is so simple, it satisfies just this simple type invariant.

We can use TLC, the TLA+ model checker, to check that \( \text{AltInvariant} \) actually is an invariant of \( \text{AltSpec} \). We must, of course, have added the definition of \( \text{AltInvariant} \) to module \textit{Alternation} (in file \texttt{Alternation.tla}) and the definitions of \textit{Output} and \( \text{CRInvariant} \) to module \textit{CallReturn} (in file \texttt{CallReturn.tla}). We then create the following configuration file \texttt{Alternation.cfg}:

\[
\text{SPECIFICATION AltSpec INVARIANT AltInvariant}
\]

We then run TLC (as described on the web). It will report that \( \text{AltInvariant} \) is indeed an invariant of \( \text{AltSpec} \).
## 1.5 Implementation

In introducing the *Alternation* specification, we said that the variable $x$ describes a discrete oscillator. This means that the specification $\text{AltSpec}$ specifies the same sequence of values for the variable $x$ as the oscillation specification does. Let $\text{OscSpec}$ be the oscillator specification, which we can write in TLA$^+$ as:

$$\text{OscSpec} \triangleq (x = 0) \land \Box [x' = (x + 1) \% 2]_x$$

Every behavior that satisfies $\text{AltSpec}$ also satisfies formula $\text{OscSpec}$. In other words, $\text{AltSpec} \Rightarrow \text{OscSpec}$ is a theorem.

How would we prove this theorem? Let’s give names to the initial predicate and next-state action of the oscillator specification:

$$\text{OsciInit} \triangleq x = 0$$
$$\text{OsciNext} \triangleq x' = (x + 1) \% 2$$

To prove that every behavior satisfying $\text{AltSpec}$ also satisfies $\text{OscSpec}$, it suffices to prove:

- Every state satisfying $\text{OsciInit}$ also satisfies $\text{OsciInit}$.
- Every step satisfying $\text{OsciNext}$ also satisfies $\text{OsciNext}$.

In other words, it suffices to prove $\text{AltInit} \Rightarrow \text{OsciInit}$ and $\text{AltNext} \Rightarrow \text{OsciNext}$. These proofs are very simple.

Instead of proving this theorem ourselves, we can let TLC verify it. To do this, we need to put the definition of $\text{OsciNext}$ into a module. We could add it to the *Alternation* module, but let’s instead write a new module that extends the *Alternation* module. We write module *AltImplementsOsc*, shown in Figure 1.4 on this page. The module also extends the *Naturals* module, which defines the operators $+$ and $\%$ used in the definition of $\text{OscSpec}$. (This isn’t necessary, since the *Naturals* module was already extended by *CallReturn*, which is extended by *Alternation*. However, we might later want to modify the *CallReturn* module, in which case it might not extend *Naturals.* ) We now create the following configuration file AltImplementsOsc.cfg
1.5. IMPLEMENTATION

SPECIFICATION AltSpec PROPERTY OscSpec

which tells TLC to check the theorem AltSpec \Rightarrow OscSpec, and run TLC.

The simple theorem AltSpec \Rightarrow OscSpec contains an important lesson. The oscillator is a system described by the single variable $x$. It is natural to think of a possible state of the oscillator as an assignment of a value to $x$, and to think of OscSpec as an assertion about sequences of possible oscillator states. Similarly, it’s natural to think of a possible state of the alternation system as an assignment of values to the three variables $alt$, $rtn$, and $x$; and to think of AltSpec as an assertion about sequences of such states. However, such thinking leads to the conclusion that OscSpec and AltSpec are assertions about different kinds of sequences: one is a sequence of values of one variable, while the other is a sequence of three variables. If this were the case, the formula AltSpec \Rightarrow OscSpec would be meaningless. How can an assertion about apples imply an assertion about oranges?

To understand what the theorem AltSpec \Rightarrow OscSpec means, we must realize that, in math, writing a formula that mentions only $x$ doesn’t imply that $x$ is the only variable in the universe. When we write mathematical formulas, we assume an infinite number of variables, like $x$, $y$, $z$, $arg$, $rtn$, $i$, $sum$, $balance$ and so on. A particular formula only mentions a finite number of those variables. A state is an assignment of values to all of those infinite number of variables. Specifications AltSpec and OscSpec are both assertions about states; AltSpec depends on the values of some variables (arg and rtn) that aren’t mentioned in OscSpec. That is, OscSpec doesn’t specify anything about the value of the variables arg and rtn, and neither specify anything about the values of an infinite set of variables including $i$ and balance. So, it’s perfectly reasonable for a formula that mentions $arg$, $rtn$, and $x$ to imply a formula that mentions only $x$. An assertion about apples and oranges can imply an assertion about oranges.

A good way to think about all this is that the values of all possible variables determines the state of the entire universe. A behavior, which is an infinite sequence of states, represents a history of the entire universe. The state of a particular system is described by the values of a finite set of variables. We sometimes call an assignment of values to that set of variables a system state. A specification of the system is a formula mentioning only those system variables, and hence its truth value depends only on the sequence of system states. But, it is still an assertion about the history of the entire universe.

We had called formula AltSpec a specification of the alternation system, but we just saw that it is also an implementation of the oscillator specification OscSpec. There is no fundamental distinction between specifications and implementations. We simply have specifications, some of which implement other specifications. A Java program can be viewed as a specification of a JVM (Java Virtual Machine) program, which can be viewed as a specification of an assembly language program, which can be viewed as a specification of an execution of the computer’s machine instructions, which can be viewed as a specification of
an execution of its register-transfer level design, and so on.

1.6 Summary

Here are the concepts we’ve introduced.

- A \textit{state} is an assignment of values to all possible variables. A \textit{step} is a pair of states. A \textit{behavior} is an infinite sequence of states; it represents a possible execution of the universe.

- A \textit{system state} is an assignment of values to the variables that describe the system.

- We write three kinds of formulas: \textit{state functions, actions, and temporal formulas}. A state function has a value in a state. A \textit{state predicate} is a Boolean-valued state function; it is true or false of a state. An action is true or false of a step. A temporal formula is true or false of a behavior. It is a \textit{theorem} if it is true of all behaviors. The only kinds of non-degenerate temporal formulas that we’ve seen so far are: (i) \( \Box P \), for \( P \) a predicate, (ii) \( \Box[A]v \) where \( A \) is an action and the \([ \ ] \) has yet to be explained, and (iii) formulas of the form \( \exists z : F \) where \( z \) is a variable and \( F \) a temporal formula.

- A system specification is a formula that is true of a behavior iff that behavior represents a legal execution of the system. For now, our specifications have the form \( \text{Init} \land \Box[\text{Next}]v \), where \( \text{Init} \) is the initial predicate, \( \text{Next} \) the next-state action, and \( v \) is a state function. We also use the \( \exists \) operator to hide variables in such a specification. Variables of a specification that are not hidden are said to be visible.

- A state predicate \( \text{Inv} \) is an \textit{invariant} of a formula \( \text{Spec} \) iff \( \text{Spec} \Rightarrow \Box \text{Inv} \) is a theorem. A \textit{type invariant} has the form \( v \in S \); it asserts that variable \( v \) has type \( S \).

- A specification \( S_1 \) implements a specification \( S_2 \) iff \( S_1 \Rightarrow S_2 \) is a theorem.

- We have written simple specifications in TLA\(^+\), checked them syntactically with the Syntax Analyzer, and checked their invariants with TLC.

1.7 Problems

\textbf{Problem 1.1} Write a specification of a 1-dimensional random walk between two walls. Let \( x \) be the current position, starting with \( x = 0 \). Each step either increments or decrements \( x \) by 1, but must leave it between 0 and 100. Write the type invariant and check it with TLC.
Problem 1.2  Explain why the oscillator specification $OscSpec$ implements the specification in problem 1.1. Write a TLA+ module asserting that it does, and use TLC to check this theorem.